Aspects of Java Program Verification

Kerry Trentelman

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

April 2006

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Kerry Trentelman
15 April 2006

For Renae.

Acknowledgements

My boundless gratitude goes to the following people and organisations.

My industrial sponsors Gemplus who, having withdrawn from supervision of my PhD after facing major cutbacks to their research laboratories, still continued to contribute to my stipend.

The Lemme team at INRIA, Sophia Antipolis, for stepping into the supervisory gap left by Gemplus. I enjoyed immensely my six month stay with the team and would like to take this opportunity to yet again apologise for my abominable French.

Dr. Marieke Huisman from INRIA, for being an absolute inspiration. It's been an honour to learn from, and work alongside, such a kind, patient and generally wonderful person.

The KeY team at the University of Karlsruhe for hosting me over a six week visit. Especial thanks go to Dr. Bernhard Beckert who, despite being ridiculously busy, still managed to generously spend time with me answering questions and collaborating on a paper.

Dr. Jeremy Dawson for Isabelle help and the reading of various manuscripts.

My supervisor, Dr. Rajeev Goré, the man with the golden brain, for his continual guidance, encouragement, support, humour and enthusiasm over these past four years. He has been such a tremendously positive force throughout my PhD, for which I am eternally grateful.

My examiners, Dr. Erik Poll, Dr. Dilian Gurov and Prof. Peter Schmitt, for their sharp and insightful comments.

Our Computer Science Laboratory administrator: the marvellous Michelle Moravec for discussions on all things film and Whovian.

My fabulous fellow students, both near and far: Pietro Abate, Nicolette Bonnette, Agnes Boskovitz, Néstor Cataño, Evan Greensmith, Charles Gretton, Kee Siong Ng, Greg O'Keefe, Cheng Soon Ong, Daniel Perovich, David Price, Edward Harrington, Leonardo Rodríguez, Tatiana Semenova, Andrew Slater and Paul Wong(as).

And lastly, for their support, advice and irregular pints down the pub, heartfelt thanks go to the Adelaide contingent: Renae Trentelman and Julian Barnett, John and Natasha Trentelman, Emily Raven and Rachel Mead, Juliette Woods and Damien Warman.

Abstract

This thesis examines aspects of Java program verification by utilising formal methods. This is a subdiscipline of software engineering where mathematically based techniques are used for the specification, development and verification of software and hardware systems. The process of program verification involves formally proving that a program does exactly what is claimed in the program specification it was written to realise. A considerable proportion of formal method techniques are devoted to the furtherance of smartcard technology.

A smartcard is a small device which stores and processes information through the microprocessor chip embedded in its plastic substrate. It may be the size of a credit card or a SIM card in a mobile phone. The microprocessor of the card executes small application programs called applets. Multiple applets can be held in the card's memory simultaneously and applets can also be downloaded onto the card post-issuance. Of the three non-proprietary smartcard operating systems: Multos, JavaCard and Windows for Smart Cards; JavaCard is by far the most popular. For several reasons JavaCard – a dialect of Java – is an ideal testing ground for the application of formal methods: (1) the language is relatively simple; (2) JavaCard's application programming interface (a set of pre-defined functions) is small, currently consisting of only 47 classes; (3) due to restrictions imposed on a smartcard's processors, the applets are quite small; and (4) programs are often used in great numbers and/or in security or privacy-critical applications, hence their correctness is deemed of significant importance.

Numerous specification languages and verification tools have been designed and developed with Java(Card) as a target programming language. Increasingly, formal method researchers are performing industrial-sized case studies on the verification of Java(Card) programs. As mentioned previously, this thesis examines particular aspects of Java(Card) program verification. We begin by extending the Java Modeling Language, JML, with temporal specifications. This extension enables users to specify the interactions between different Java(Card) program components which ordinarily cannot be specified in JML. We next propose a method to factorise the verification of temporal properties for multi-threaded Java programs over their different threads. This method lightens the verification burden placed upon tools by multi-threaded programs. Following this, we examine how properties of relations (such as transitive closure, finiteness and generatedness) are described in various logics, with emphasis on extensions of first-order logics. Because these logics are often implemented within specification languages in an *ad hoc* fashion, we feel it necessary to clarify a number of issues. Finally we look at proving the correctness of a verification tool, ensuring that the rules of a calculus – implemented within the KeY verification tool – are sound with respect to the Java(Card) programming language.

Contents

Chapter 1

Introduction

In our technological age we find that the complexity of software and hardware systems is increasing all the time. This increase in complexity brings about greater margins for error whereby faults become commonplace. Faults – commonly termed "bugs" – arise from a failure to anticipate particular interactions among the components of a system, or from a mismatch between the system and its environment. Such faults often have the potential to cause financial ruin or even endanger people's lives. One of the main goals of software engineering is the construction of systems that operate reliably despite their complexity. The approach of formal methods is used in an attempt to achieve this goal. The loosely coined term "formal methods" describes a collection of mathematically based techniques used for the specification, development and verification of software and hardware systems [147]. Their purpose is to reveal inconsistencies and ambiguities within safety-critical and commercial systems. If used during the early developmental phases, formal methods can often reveal design flaws within software systems which might otherwise have been detected only during costly testing and debugging phases. With most large projects spending over 50 percent of their development time on debugging, there are tremendous opportunities for using formal method techniques to improve upon the software and hardware system design process [143].

A formal method's mathematical basis is provided by a specification language. Different formal method approaches are commonly associated with different specification languages, and the languages themselves are often intended for different purposes [125]. Many languages are based on an underlying formal logic. A specification of a computational system – described in terms of a specification language – captures particular assumptions about the context in which the system is to operate and also expresses the required properties of the system. It is important that specifications are both unambiguous and consistent [147]. A specification is unambiguous if and only if it has exactly one meaning, whereas a specification is consistent if we are unable to derive anything contradictory from it. The procedure of formal verification establishes – by means of formal deduction – that a system satisfies its specification. Two established approaches to formal verification are model checking and theorem proving.

Model checking This approach involves building a finite model of a system and then checking that a desired temporal property holds over the model [36]. The check is performed exhaustively over the entire state space. A major problem faced by the model checking community is that of state explosion; many system components interact in parallel, hence the size of the model is exponential with respect to the size of each component. Thus it is vital to devise

algorithms, *i.e.* model checkers, that can handle large search spaces. Unlike theorem proving, model checking is completely automatic and very fast. It can also be used to check lightweight, *i.e.* partial specifications, finding common errors quickly and efficiently. Runtime checking and static checking are the two principal techniques used for the verification of lightweight specifications. Runtime checking checks properties during the program's execution, whereas static checking finds potential runtime problems at compile time. Two examples of model checking tools include ESC/Java [51] an "extended static checker" discussed in Section 2.2.1, and the Java PathFinder [87] which is used to verify the multi-threaded operating system of NASA's Deep-Space 1 space-craft. The PathFinder translates Java into PROMELA, the programming language of the Spin model checker [134].

Theorem proving This is a technique whereby both the system to be verified and its desirable properties are described as formulae of some logic [36; 126]. The logic is given by a calculus which defines a set of axioms and a set of inference rules. Theorem proving is the machine-assisted process of finding – *via* the calculus – a proof of a property of (the logical description of) the system. The machine assistance comes in the guise of a computer program, or "theorem prover", which uses search and heuristics in order to help find the proof. Note that proofs can be found inductively. Steps in the proof are made by utilising the axioms and rules, and possibly derived definitions and intermediate lemmata. The popular theorem prover Isabelle/HOL [76] features prominently throughout this thesis; it is introduced in Section 5.3.

Theorem provers and model checkers are often more effective in combination. Large sections of less complicated specifications can be model checked, whereas theorem proving can be applied to the more complex parts of the specifications. One of the great strengths of formal methods is that it permits the analysis of all possible behaviours of a system. This is highly desirable since we are only able to state conclusively that a system will not fail after we have exhaustively explored that system's behaviour. We follow Rushby's argument in [125] in order to explain how this is possible: formal methods allows us to (1) characterise particular behaviours of a system by mathematical expressions called formal specifications, and (2) deduce further behaviours of that system by applying formal deduction. Eventually one can compose the complete behaviour of the system, covering all possible end-to-end behaviours without having to enumerate them explicitly.

Formal methods has had its fair share of critics, particularly during its formative years: notations were said to be too cryptic; techniques failed to scale-up; tool support was inadequate or difficult to use; and the verification of even small systems was painfully slow. Gradually however a sea-change has come about. With the advancements made in this field, verification techniques such as model checking and theorem proving have now gained acceptance by the industry, complementing traditional techniques such as testing and simulation. Increasingly, researchers are performing more industrial-sized case studies; the majority of these involve the formal verification of application programs designed to run on smartcards.

Smartcards A smartcard is a small device which stores and processes information through the microprocessor chip embedded in its plastic substrate [32]. It may be the size of a credit card or a SIM card in a mobile phone. The microprocessor of the card executes small appli-

cation programs called applets. Multiple applets may be held in the card's memory simultaneously. Moreover, applets can be downloaded onto the card after it has been issued, allowing services to be updated or new services to be added. The current generation of smartcards has an eight-bit processor, 16KB read-only memory, and 512 bytes of random-access memory [89]. In order to begin a transaction, a smartcard is either inserted or brought sufficiently close to a smartcard reader. The card communicates with a reader by exchanging application protocol data units, or APDUs. These contain either command or response messages. For example a reader may order the card to select a particular applet, or the card may respond with an error message. The language JavaCard is commonly used for programming smartcards. It is a particular dialect of Java and was first introduced by the smartcard companies Schlumberger Sema and Gemplus [146]. The JavaCard language features prominently throughout this thesis; however before we examine it further, we will discuss JavaCard's superset language Java.

Java Java is one of the most popular object-oriented programming languages in use around the world today. Originally developed by James Gosling *et al.* at Sun Microsystems, it was publicly released in 1994 and gained prominence in 1995 when Netscape announced they would provide support for it in their Navigator browser [146]. Object-oriented programs are labelled such because they are comprised of a number of interacting objects. These can be considered as self-contained bundles of code describing the object's behaviour, as well as including the data the object itself manipulates. Each object is described by a class which consists of both field and method declarations. Classes are by nature modular and hence can be re-used in different applications. Java – and typically most other object-oriented programming languages – comes with a set of commonly used functions called an Application Programming Interface, or API. The Java API provides Java with all of its basic behaviour and can be considered the "building blocks" of the language. Formal verification of the methods within the API is highly desirable, increasing the reliability of the programs based upon them; Chapter 2 of this thesis addresses this issue in detail.

Java source code is compiled into bytecode which is designed to run on a virtual machine [6]. Bytecode is simplified machine instructions which are platform independent, meaning that they can be executed by a virtual machine on any system that supports the Java programming language. A virtual machine is a program written in the native code of the host hardware which translates Java bytecode into usable code on the hardware. When bytecode is loaded into a virtual machine it is first checked by a bytecode verifier. The verifier ensures that the bytecode respects certain security-related properties or constraints related to its syntax, its behaviour, and any potential inter-dependencies between the code. The virtual machine has the ability to control the actions that the loaded bytecode is permitted to take. This is essential if untrusted bytecode from remote sources is to be safely executed. Typically, such untrusted code comes in the form of applets. Applets are Java programs which can be included in an HTML page. When a Java-enabled browser is used to view a page that contains an applet, the applet's bytecode is downloaded from a remote HTTP server and executed by the browser's virtual machine within a highly restricted "sandbox". This protects the user from malicious code, since the set of operations permitted in the sandbox is limited. Applets may come with a certificate that digitally signs them as "safe" thereby giving them permission to leave the sandbox and access the local file system and network.

JavaCard As mentioned previously, JavaCard is a particular dialect of Java used to program smartcards. Of the three non-proprietary smartcard operating systems: Multos, JavaCard and Windows for Smart Cards; JavaCard is by far the most popular. JavaCard supports the features of Java that are well suited for writing programs for smartcards and meanwhile preserves Java's object-oriented capabilities. It also supports concepts not available in Java, such as the transaction mechanism discussed in Section 2.5. The features of Java that JavaCard does not support include: large primitive datatypes such as `long`, `double` and `float`; multi-dimensional arrays; dynamic class loading; garbage collection; threads; and object cloning.

For several reasons JavaCard is an ideal testing ground for the application of formal methods: (1) the language does not include threads, floating point numbers, nor multi-dimensional arrays, and is in itself relatively simple; (2) JavaCard's 2.1 API is small, currently consisting of only 47 classes; (3) due to restrictions imposed on a smartcard's processor, the applets are quite small; and (4) programs are often used in great numbers and/or in security or privacy-critical applications such as banking, telecommunications and health care, hence their correctness is deemed of significant importance.

VerifiCard The VerifiCard project – spanning January 2001 to September 2003 – brought together five academic and two industrial partners in order to work on the correctness of components of the JavaCard platform, and also of individual applications [142; 81]. The partners were assigned different focus areas, for example: the Technical University of Munich worked on the operational and axiomatic semantics of JavaCard; the University of Nijmegen and the University of Hagen worked on a complete formal specification of the JavaCard API; and the French National Institute for Research in Computer Science and Control (INRIA), along with the smartcard company Gemplus, worked on the formalisation of the JavaCard virtual machine and a certified byte code verifier. At the project's end the consortium had provided the most comprehensive formalisation of the JavaCard platform to date. Furthermore, it had provided several tools and techniques for the specification and verification of JavaCard applets. These tools and techniques were tested on realistic JavaCard applets provided by the industrial partners. A final report can be found at [142]. The work outlined in Chapter 2 of this thesis was conducted as part of the VerifiCard project.

This thesis examines particular aspects of Java(Card) program verification. The remainder of this thesis is outlined in the following paragraphs.

Chapter 2 In this chapter we propose an extension of the specification language JML (Java Modeling Language) with temporal specifications. The extension is inspired by the specification "patterns" used within the Bandera project and is especially tailored to specify properties of Java(Card) programs. Following the tradition of JML, the extension has been designed to be simple, easy and intuitive to use for software engineers. We show how the JML extension can be used to specify temporal aspects of the JavaCard Application Programming Interface (API), and later discuss a semantics for the extension. We also show how we can translate a subset of the extension back into standard JML, thereby allowing for the re-use of existing verification techniques. A trace-based semantics is given for the "new" part of the specification language.

Chapter 3 Here we propose a method to factorise the verification of temporal properties for multi-threaded Java programs over their different threads. Essentially the method involves showing that some of the threads establish the property, while the other threads do not affect it. The method is fine-tuned by identifying necessary preservation conditions for each property. As a specification language, we again use the specification patterns developed by the Bandera team. For each specification pattern a decomposition rule is proposed. We show the soundness of each rule using the pattern mappings as defined for linear temporal logic. The proofs have all been formalised using the theorem prover Isabelle/HOL.

Chapter 4 In this chapter we give an overview of the different ways properties of relations – such as transitive closure, infiniteness and generatedness – can be expressed in extensions of first-order logic, *i.e.* transitive closure logic, fixed-point logic and first-order dynamic logic. (Such properties cannot be expressed in first-order logic alone.) Within the chapter we also discuss which of these extensions already are – or in fact should be – implemented within object-oriented specification languages. We pay particular attention to the Object Constraint Language, OCL, and the Java Modeling Language, JML.

Chapter 5 Every methodology for the verification of Java programs involves firstly transforming the (informal) Java language specification into some formal specification. Since there is no way this transformation can be formally proved correct, the best we can do is to compare independently obtained formalisations given in different formal specification languages. This chapter provides such a comparison. Developed at the University of Karlsruhe, KeY is an augmented commercial CASE (Computer-Aided Software Engineering) tool with specification and deductive verification functionalities. KeY implements a sequent calculus called JavaCard DL, which has been designed to capture the semantics of JavaCard. The chapter discusses a case-study into proving JavaCard DL sound using the independent Bali operational semantics for Java (which has been embedded in the theorem prover Isabelle/HOL). Rather than taking a foundational approach by embedding the entire JavaCard DL semantics directly into a theorem prover, we instead translate each rule and prove its soundness *via* Bali. We examine both JavaCard DL and Bali approaches, prove three pivotal rules sound, and argue whether the method is useful in proving the relative correctness of JavaCard programs overall.

Chapter 6 Here we draw conclusions and discuss the principal contributions of this thesis.

Together these chapters address three issues of particular importance to the formal methods community: making specification languages more expressive; lightening the verification burden; and ensuring the correctness of verification tools overall. Both Chapter 2 and Chapter 4 are concerned with making specification languages more expressive. In Chapter 2 we extend the Java Modeling Language, JML with temporal specifications. This enables a user to specify the interactions between different Java program components which ordinarily cannot be specified in JML, *i.e.* that a given property will hold between two particular called methods. In Chapter 4 we examine how properties of relations are described in various logics, with emphasis on extensions of first-order logics. Since many specification languages lack the (much needed) ability to describe properties of relations, it is a worthwhile exercise to investigate the

means by which such properties can be expressed and how these means can be implemented within various languages.

We find that one of the downsides of making a specification language more expressive is that we often make the verification task more complex. Consider, for example, the difficulties faced when verifying liveness properties, *i.e.* "eventually" something good will happen; such properties can only be violated at infinity. The complexity of the verification tasks are exacerbated further when it comes to verifying specifications over multi-threaded programs; here we encounter the state explosion problem where it becomes necessary to take into account all the possible interleavings of all the different threads. Chapter 3 proposes a method to alleviate this problem. The method can be used in conjunction with other techniques such as abstraction, slicing and atomicity checking (all of which have been designed to reduce the proof burden caused by state explosion). Furthermore the method builds upon the specification language developed in Chapter 2.

Chapter 5 looks at proving the correctness of a verification tool itself, ensuring that the rules of a calculus – implemented within the tool – are sound with respect to the Java programming language. Such soundness proofs are highly desirable, since if a verification tool cannot accurately model its target programming language, then there is very little one can conclude about the correctness of a program's verification.

Published material Much of this thesis is joint work with others and a considerable amount has been published elsewhere. Chapter 2 – which proposes adding temporal specifications to JML – expands upon a paper written with Dr. Marieke Huisman [140]. In contrast to the paper, this chapter specifies temporal aspects of the entire JavaCard API (rather than just the transaction mechanism) and describes our extension's translation back into standard JML in much greater detail. Chapter 3 – which looks at factorising temporal specifications – also expands upon a paper written with Dr. Huisman [73]. Deviating from the paper, we prove a number of interesting results arising from our formalism. We also describe our program model and factorisation method in greater detail. Chapter 4 – which examines the implementation of properties of relations within object-oriented specification languages – builds generally upon a paper written by the author in collaboration with Dr. Bernhard Beckert [14]. Chapter 5 – which looks at proving a number of tool-implemented rules sound – expands upon a paper written by the author [139]. As opposed to the paper, we delve more deeply into the Bali and JavaCard DL calculi so that the chapter is self-contained.

Chapter 2

Extending JML with temporal specifications

This chapter proposes an extension of the Java Modeling Language (JML) by incorporating temporal specifications. The extension is inspired by the patterns and specification language used within the Bandera project, and is especially tailored to specify properties of Java(Card) programs; for example, it allows the exceptional behaviour of methods to be specified. In the tradition of JML, the extension is designed to be simple, easy and intuitive to use for software engineers. The chapter expands upon an existing paper written with Marieke Huisman [140]. In the original paper we used the JML extension to specify temporal aspects of JavaCard's transaction mechanism. We also outlined how to translate a subset of the extension back into standard JML, thus allowing the re-use of existing verification techniques for JML. Here, notably, we specify temporal aspects of the entire JavaCard API and describe our extension's translation back into standard JML in much more detail.

2.1 Motivation

Although the feasibility of program verification is now acknowledged, it is still labour-intensive and complex, requiring an appropriate specification language and an understanding of the underlying semantics. (See [69; 71] for some examples of verification of Java programs using theorem proving.) Design by Contract lead the first advance; it being a method promoting the use of assertions to incorporate specification information into the program code itself [106]. Eiffel was the first programming language based on Design by Contract, its assertions describing the code's implicit contracts and specifying requirements such as: preconditions that a client must meet before a method is invoked, postconditions that a method must meet after it executes, and class invariants that must be preserved by each method [105]. Following this approach, several specification languages designed for Java have been developed, among them the Java Modeling Language, or JML [98; 97; 99]. However, verifications of Java programs using specification languages based on Design by Contract can only determine the correctness of a program's functional behaviour. (Functional specifications describe the specific functions, tasks or behaviours the program must support, whereas non-functional specifications are constraints on various attributes of these functions or tasks.) The non-functional behaviour of a program cannot be specified; this is due to the nature of Design by Contract which does not

allow the specification of constraints on interactions between different program components.

Recently, in the field of model checking, this omission is being addressed. (See [40; 66] for examples of model checkers applied to Java.) A drawback of these approaches is that the specifications are often given in a complicated logic which makes them difficult to understand for many Java programmers. Also the specifications are usually given separately – they are not part of the program – in contrast to *e.g.* Eiffel or JML specifications. These different specification techniques should be unified in order to get a closer integration between functional and non-functional (in particular, temporal) specifications.

Java with Assertions, or Jass, is a first attempt to bridge this gap. Developed at the University of Oldenburg, Jass allows Java classes to be annotated with specifications in the form of assertions [10; 86]. In particular, Jass features trace assertions which are used to monitor the ordering of method invocations and returns. Trace assertions – whose semantics are based on Communicating Sequential Processes, CSP [68] – can be used to specify the order in which methods can or must be invoked, and also the conditions under which a method can be invoked. However, Jass trace assertions cannot be integrated with other Jass specifications, hence a specification stating *e.g.* that after a particular method call a certain variable should always be positive, is not allowed.

JML was originally designed by Gary Leavens *et al.* at Iowa State University in 1998. Having spawned a much larger community of users and tool developers who are now actively involved in its development, JML has since become the standard specification language used for verification of Java programs within the academic community. JML is used to specify Java classes and interfaces [98; 97; 99]. Many of the standard assertion features of JML are similar to Jass. However JML's ability to handle interfaces and abstract classes, and its feature of model variables – described in more detail in the next section – makes it much more expressive. (In fact, the designers of Jass are currently considering switching to JML as an assertion language.) We therefore propose an extension of JML with temporal logic which is easier to understand than Jass trace assertions and which can be used to specify the temporal behaviour of interactions between different objects in a Java(Card) program.

In order to develop a better understanding of what kind of specification constructs are necessary for our extension language, we looked at the JavaCard Application Programming Interface (API) [88]. As discussed in the introduction, JavaCard is a particular dialect of Java used for programming smart cards [89]. Because of JavaCard's relative simplicity (*cf.* Java) as well as the security and privacy-critical nature of smartcard technology, JavaCard is an ideal testing ground for the application of formal methods.

The temporal extension of JML presented in this paper has been designed with the following goals in mind:

- the language should be intuitive and easy to understand for software engineers familiar with Java(Card);
- the language should be tailored to specify properties about Java(Card) programs;
- the language should be integrated with standard JML, *i.e.* it should be possible to use existing JML expressions in the temporal logic formulae;

- the language should provide all specification constructs that have been identified as important by the specification patterns project and which are relevant for Java; and
- the language should have a clear semantics and appropriate verification techniques.

A subset of the specifications allowed by our language extension can be translated back into standard JML specifications; they can therefore be verified using standard verification techniques for JML, as advocated in *e.g.* the LOOP project [83]. This subset describes safety properties. Specifications which cannot be translated back into standard JML are liveness properties. Intuitively, a safety property specifies that "bad things" do not happen on all executions of a system, whereas a liveness property specifies that "good things" eventually happen on all executions of a system. More formally – as defined by Alpern and Schneider [5] – a property is a safety property if and only if every infinite sequence of states that does not satisfy the property contains a finite prefix such that no infinite sequence obtained by adding an infinite suffix to this prefix satisfies the property. A property is a liveness property if and only if for every finite sequence we can find an infinite suffix, so that the resulting infinite sequence satisfies the property. This chapter describes the semantics of both safety and liveness properties, but it does not present appropriate verification techniques for liveness properties; this has recently been addressed in [15] and is discussed below.

Related work For the extension of JML we are inspired by the Specification Pattern project, a branch of the Bandera project [9]. The overall goal of the Bandera project is the development of a tool (Bandera) which automatically extracts abstract mathematical models based on specified properties from Java source code. The tool can then render these models in the input language of several different model checking tools. Bandera employs program analysis, abstraction and transformation techniques in order to extract the finite-state models [40]. Specifications of Java source code are made using the Bandera Specification Language, BSL, which implements so-called specification patterns. A specification pattern is a language independent set of commonly used specification constructs for finite-state verification, with mappings into different temporal logics. Because of this implementation, BSL is independent from the source code it specifies [123; 41]. However BSL does not allow properties of exceptions to be specified; this is where it differs from JML and our work.

Members of the Security of Systems (SoS) group – formerly known as the LOOP group – at the University of Nijmegen [132] have written specifications for the JavaCard API in JML [90; 104; 117]. A goal is to eventually verify these specifications with respect to the reference implementation of the API using the LOOP tool (discussed in Section 2.2.1). In several specifications, temporal aspects – such as the order of method invocations – have been specified using model fields, simulating a state automaton. (Section 2.5 shows how similar specifications look in our JML extension.) However some temporal specifications have not been given at all, since standard JML is inadequate to specify these directly.

Our work here has inspired Bellegarde, Groslambert *et al.* who have recently proposed a way to verify liveness properties expressed using our extension language [15]. The verification of a class's liveness property is decomposed into two tasks. The first of these is showing that the class is run in an "ideal" environment, *i.e.* an environment that calls all methods sufficiently often. If such is the case, then the liveness property may be established. The second task is

checking whether the class itself establishes the liveness property. We discuss their approach in greater detail in the latter part of Section 2.6.

This chapter is outlined as follows: in Section 2.2 JML is described in detail and example specifications are given; Section 2.3 outlines the Bandera specification patterns; Section 2.4 formally describes the semantics of our specification language. Section 2.5 presents our proposed specifications for the temporal aspects of JavaCard; Section 2.6 discusses how we translate our specification language back into standard JML; and finally in Section 2.7, we draw conclusions and discuss future work.

2.2 The Java Modeling Language

JML has been designed to be easy to use for Java programmers with little experience in logic. Remaining faithful to Java syntax and semantics, it allows class invariants and pre- and post-conditions for methods and constructors to be written within the program code. Specifications are formulated by making use of (side-effect-free) boolean Java expressions; they are written as Java comments, following `//@` or enclosed between `/*@` and `*/`. The additional `@` notifies the JML tool that it is a JML specification rather than an ordinary comment. The original JML tool is a pre-compiler designed to translate specified programs into Java programs which explicitly monitor assertions at runtime. Specification violations that are found throw Java exceptions. Since JML's conception several more tools have been developed which use JML as an input specification language. Some of these are discussed in Section 2.2.1. For a more extensive overview of JML tools and applications, see [24]. A simple JML method specification is described below. Note that all examples in this section are taken from [97; 99].

```
public class IntMathOps {                                          1
                                                                   2
   /*@ public normal_behavior                                      3
     @   requires y>=0;                                            4
     @   assignable \nothing;                                      5
     @   ensures 0<=\result && \result<=y                          6
     @        && \result*\result<=y                                7
     @        && 0<=(\result+1) ==>                                8
     @                y<((\result+1)*(\result+1));                 9
      */                                                           10
   public static int isqrt(int y)                                  11
   {                                                               12
     return (int)Math.sqrt(y);                                     13
   }
}
```

The specification describes a Java class `IntMathOps` (declared in line 1) that contains a method `isqrt` (declared in line 11). From the Java code we can see that the class and method declarations are `public`, the method is `static`, the method's return type is `int`, and the method takes one integer `y` as argument. The behaviour of the class is specified in the JML annotations

between `/*@` and `*/`. The first annotation (declared in line 3) says that the specification is a public, normal behaviour specification. Specifications can be given different privacy levels such as `public`, which is intended for the use of clients, and `protected`, which is intended for subclasses.

The keyword `normal_behavior` tells the JML tool that when the precondition is met, the method must return normally without throwing an exception. The precondition is contained within the `requires` clause (line 4). In our example it states that the integer `y` must be greater than or equal to zero before the method `isqrt` is called. Following the principles of Design by Contract, if the precondition is not met, then the method is not obligated to comply with the rest of the specification. The `assignable` clause (declared in line 5) describes frame conditions; only the locations (*i.e.* fields of objects) named in the clause can be assigned to during the execution of the method. The keyword `\nothing` is often used. It has the meaning that the method may not assign to any locations that are visible outside the method and which existed before the method started execution; it may however modify its own, local variables.

The normal postcondition is contained in the `ensures` clause (line 6). If the precondition is true, then the method must terminate normally in a state satisfying the (normal) postcondition. This is comparable to a total correctness formula in Hoare logic. In our example the postcondition states that the result is an integer approximation to the square root of `y`. Note that the keyword `\result` denotes the value returned by the method. The postcondition is comprised of four conjuncts, three of which require further explanation. The first two conjuncts (line 6), as discussed in [99], are needed to ensure that the approximation does not result from overflow; this arises in Java when multiplying `int` values. The fourth conjunct (spanning lines 8-9) states that if the successor of the result is non-negative, then the successor of the result squared is strictly larger than `y`. The implication is necessary because if the result is larger than 46340, the result plus one squared will become negative due to integer overflow. The JML logical operators are simply the Java logical operators. Along with the binary boolean operators `&` and `|` are the conditional operators `&&` and `||`. These operators are such that they evaluate their right-hand operand only if the value of the expression has not already been determined by the left-hand operand [32].

Exceptional behaviour specifications may also be written. The `exceptional_behavior` keyword tells the JML tool that when the precondition is met, the method must throw an exception. A variation on the normal and exceptional behaviour specifications is the behaviour specification. The `behavior` keyword tells the JML tool that when the precondition is met one of three things can happen: (1) if the method terminates normally, then the normal postcondition holds; (2) if the execution of the method terminates by throwing an exception of the type stipulated in brackets within the `signals` clause, then the exceptional postcondition (contained in the `signals` clause) holds; and (3) if the method fails to terminate, *i.e.* it loops forever or exits without returning or throwing an exception, then the predicate listed in the `diverges` clause holds. These specifications can be translated into formulae of an extended Hoare logic dealing with abrupt termination outlined in [70; 83]. We present an example `behavior` specification below.

```
public abstract class Diverges {

    /*@ public behavior
      @    diverges true;
      @    assignable \nothing;
      @    ensures false;
      @    signals (Exception) false;
       */
    public static void abort ();
}
```

This specification describes an abstract class `Diverges` that contains a method `abort`. The class and method declarations are `public`. The method is `static`, takes no parameters and does not return any values, *i.e.* it is `void`. The public, behaviour specification has an implicit `requires` clause with normal precondition `true`. Hence the method `abort` can always be called. Because of the predicate `true` in the `diverges` clause the method may not always return to the caller, thus it may always fail to terminate. As before the `assignable` clause dictates that the method may not assign to any locations. Because of the normal and exceptional postconditions `false` listed in the respective `ensures` and `signals` clauses, the method may never return normally nor throw an exception of type `Exception`. Thus the method is legally bound to not return to its caller.

Normal behaviour, exceptional behaviour, and behaviour specifications are all examples of heavyweight specifications. In contrast lightweight specifications may be given. Within lightweight specifications the absence of any behaviour keyword tells the JML tool that the specification may be incomplete: the user has specified only what is of particular interest to his or herself. Omitted clauses in a specification carry different defaults according to the weight of the specification. Defaults are given in Figure 2.1. Defaults for lightweight specifications using the `\not_specified` keyword are such that no assumptions can be made about the omitted clause. The keyword `\everything` used in a heavyweight `assignable` clause has the meaning that all locations can be assigned to. Note that we have not discussed all possible JML clauses here; we instead refer the interested reader to [99; 97]. It is worth highlighting that lightweight, normal and exceptional behaviour specifications can all be "desugared" into behaviour specifications. Syntactic sugar is a term used to describe the additions to the syntax of a computer language which do not affect its expressiveness, but instead make it "sweeter" (or easier to comprehend) for a human reader. For example, a normal behaviour specification in JML is simply syntactic sugar for a behaviour specification with the implicit clauses `signals (java.lang.Exception) false` and `diverges false` (the default clause), which rules out the method either throwing an exception or failing to terminate.

We conclude this section by discussing a number of aspects of JML that are of particular relevance to the work presented here. First are expressions of the form `\old(E)` which can be used within (exceptional) postconditions. Such expressions denote the value of the expression `E` evaluated in the pre-state of the method. Second are the so-called model and ghost fields which are used to get a higher level of abstraction in the specifications. They are typically used to represent the internal state of an object in an abstract way. An extension of Hoare's

omitted clause	lightweight	heavyweight
`requires`	`\not_specified`	`true`
`diverges`	`\not_specified`	`false`
`assignable`	`\not_specified`	`\everything`
`ensures`	`\not_specified`	`true`
`signals`	`(Exception) \not_specified`	`(Exception) true`

Figure 2.1: Omitted clause defaults

data abstraction technique [67], these fields are declared using the keywords `model` and `ghost` and are only allowed to exist within a specification. For example, a typical specification of an object `BoundedThing` [97] has two model fields: `size` and `MAX_SIZE`, denoting the size and the maximum size of the `BoundedThing`, respectively. The `represents` clause relates the value of a model field to the concrete fields it abstracts. In the specification of an interface for a bounded stack which extends `BoundedThing`, the clause `//@ represents size <- theStack.length();` relates the model field `theStack` to `size`. This says that the value of `size` is `theStack.length()`. In contrast, a ghost field does not have a value determined by a represents clause. To change the value of a ghost field a `set` clause is used which specifies an "assignment" to the field. In addition, both model and ghost fields can be initialised by specifying an `initially` clause. For example the specification `//@ initially size == 0;` specifies that `size` is initialised to 0. Furthermore, the keyword `instance` is used to declare a non-static ghost or model field.

Lastly, there are specification constructs which describe the behaviour of a whole class. Examples of these are invariants and constraints. An `invariant` clause declares those properties that are true in all publicly visible, reachable states of an object, *i.e.* for each state that is outside of a public method's execution. An invariant is supposed to be established by the class constructors and to be preserved by each (public) method. For example, the specification `//@ invariant 0 <= size && size <= MAX_SIZE;` means that the value of `size` is always bound between 0 and `MAX_SIZE`. Within a method's execution an invariant may be broken, but before the method terminates the invariant has to be re-established, even when the method terminates exceptionally. A `constraint` clause relates the pre-state and the post-state of every method, restricting how the variables may be changed by a method. For example `//@ constraint MAX_SIZE==\old(MAX_SIZE);` specifies that the value of `MAX_SIZE` cannot change, since the value in a method's post-state, `MAX_SIZE`, must be always equal to the value in its pre-state, `\old(MAX_SIZE)`. Within a constraint clause it is possible to explicitly list the methods which must respect the constraint; if no methods are listed, then all methods of the class must respect the constraint.

2.2.1 JML tools

Because users are often reluctant to adopt new verification tools that involve having to learn new specification languages, it is indeed advantageous that a number of tools now support JML. In [21] the authors advocate that the usage of different tools is complementary: large

sections of less complicated specifications can be checked automatically, whereas more precise verification methods can be applied to the more complex parts of the specifications. Below we discuss some of the tools currently available for the specification type checking, runtime debugging, static analysis, or verification of JML annotated programs.

ESC/Java Originally developed at the Compaq Systems Research Center, ESC/Java – which is now open source – implements what it calls "extended static checking" [54; 100]. It is an "extension" in that it handles more than just type checking. The tool is fully automated and can check reasonably simple assertions and common Java programming errors such as indexing an array out of bounds or null-pointer dereferencing. The designers have taken a utilitarian approach, deliberately making it neither sound nor complete. It is believed that by keeping ESC/Java lightweight, *i.e.* by not requiring full functional specifications, the number of program errors it will find will be maximised. A drawback of the tool is that it is not fully compatible with JML. It cannot support `constraint` or `assignable` constructs, nor can it support model variables. Fortunately, these problems have been addressed by the recent release of ESC/Java version 2 [38; 51]. ESC/Java 2 is fully compatible with JML.

LOOP The University of Nijmegen's Logic of Object-Oriented Programming tool (LOOP) translates a JML annotated Java program into theory files for the theorem prover PVS [16; 84; 115]. These files are based on a hand-written "semantic prelude" which defines the core semantics of Java and JML and also defines the machinery needed, in the form of PVS theories and lemmata, to support the actual program verification. This verification is done interactively in PVS and proceeds *via* a special Hoare logic for Java [70] in combination with a weakest precondition calculus [79]. (A formal proof of the soundness of the programming logic has been given in PVS. Moreover, the weakest precondition calculus has been proven correct with respect to the underlying Java semantics.) Of course interactive theorem proving is very labour intensive, but this method allows for the verification of much more complicated properties than those handled by *e.g.* ESC/Java.

Jack The Java Applet Correctness Kit (Jack) – originally developed at Gemplus, now at INRIA Sophia Antipolis – sits somewhere between ESC/Java and LOOP. The Jack tool – which is not publicly available – implements a fully automated weakest precondition calculus that generates proof obligations from JML annotated Java programs [25]. Currently, proof obligations can be generated for the (almost fully automatic) prover developed within the B method [1], the Simplify theorem prover [45] and the Coq proof assistant [17]. Unlike LOOP, Jack does not require its users to have expertise in the use of a theorem prover, instead it hides the complexity of the underlying concepts beneath a user-friendly interface. This interface provides a graphical view of the proof obligations, presenting them as execution paths within the program and highlighting the relevant source code. Furthermore, Java and JML-like notation is used when presenting goals and hypotheses.

Krakatoa Developed at INRIA Futurs and the Université Paris-Sud, the Krakatoa tool [93; 102] translates JML annotated Java into an input language to be read by a tool called Why [145].

Why is a stand-alone verification conditions generator and acts as a back-end for other verification tools. It takes as input ML programs annotated in a Hoare logic style and produces proof obligations using its own weakest precondition calculus. Proof obligations may be generated for the proof assistants Coq, PVS and HOL, amongst others. In Krakatoa's case, proof obligations are generated for Coq and are then proven interactively.

KeY Originally developed at the University of Karlsruhe – now a joint project of the University of Karlsruhe, Chalmers University of Technology and the University of Koblenz – the KeY system is a commercial Computer Aided Software Engineering (CASE) tool augmented with specification and deductive verification functionalities [2; 91]. KeY uses the Unified Modeling Language UML for visual modelling of designs and specifications, along with OCL for specifying constraints and other expressions attached to the models [141]. KeY implements a sequent calculus called JavaCard DL, which has been designed to capture the semantics of JavaCard. In its most recent version (0.99) KeY provides a JML interface [50; 49]. The KeY tool and the JavaCard DL sequent calculus is discussed in detail in Chapter 5.

Also worth mentioning briefly here are the Chase and Daikon tools. Developed at INRIA Sophia Antipolis, Chase is designed to check JML `assignable` clauses [28; 26]. Notoriously difficult to specify, Chase performs syntactic checks on these clauses, catching many common specification mistakes. The Daikon invariant detector was developed by the Program Analysis group at MIT and assists in creating specifications [43]. It does this by observing the runtime behaviour of a program and dynamically detecting likely invariants.

A number of case studies have been conducted which apply JML tools to JavaCard. Of these, the most interesting is the electronic purse case study which was verified in 2002 using ESC/Java at INRIA Sophia Antipolis [27] and the LOOP tool at the University of Nijmegen [22]. The electronic purse was designed by the smartcard company Gemplus [58]. Both test cases proved instrumental in motivating the development of the Jack tool. The first non-trivial applet, comprising several hundred lines of code, has been recently verified using the LOOP tool and PVS theorem prover (the applet was provided by Schlumberger Sema) [80; 82]. A case study has also recently been conducted on a Gemplus banking application [20]. Its partial JML specification was used as a test oracle (*i.e.* it was used to generate the expected outcomes of a test) in order to determine whether JML can be integrated with classical testing tools. The case study describes issues associated with using JML in a testing context. It is an encouraging sign that actual verification of smartcard applets is becoming feasible. All three verifications relied upon the JML specifications written for the JavaCard API as part of the LOOP project [90; 104; 117]. According to [24], several other projects are currently underway using ESC/Java and LOOP to verify JavaCard applets; these however are all subject to non-disclosure agreements.

2.3 Specification patterns

Influenced by the notion of design patterns, the specification pattern approach to finite-state verification was first proposed by Matthew Dwyer *et al.* of the Specification Pattern project at

Kansas State University, a branch of the Bandera project [46]. Specification patterns describe some aspect of a program's behaviour and provide expressions of this behaviour in common formalisms such as linear temporal logic, LTL, or computational tree logic, CTL. (See *e.g.* [48; 74] for a description of these formalisms.) Patterns "capture" the distilled knowledge and experience of expert specifiers in temporal logic; users need not have an in-depth knowledge of the underlying semantics of the specification language they are using, they just need to look up the relevant pattern to match the particular requirement being specified. As explained in the introduction to this chapter, the specification patterns form the basis for BSL, the input language for the model-checker front-end tool Bandera [40].

Patterns include: *occurrence* patterns such as universal, existence, absence and bounded existence; and *ordering* patterns such as precedence and response. Each pattern has a scope which gives the extent of the program execution over which the pattern must hold. The intents of the patterns mentioned are given as follows:

- *universal:* a given state or event occurs throughout a scope
- *existence:* a given state or event must occur within a scope
- *absence:* a given state or event does not occur within a scope
- *bounded existence:* a given state or event must occur at most/at least/exactly n times within a scope.
- *precedence:* a state or event must always be preceded by another state or event within a scope
- *response:* a state or event must always be followed by another state or event within a scope

There are five main scopes, describing time intervals which are closed at the left and open at the right.

- *globally:* throughout the entire program's execution
- *after:* the execution after a given state or event
- *before:* the execution up to a given state or event
- *between:* any part of the execution between two designated state or events
- *after-until:* the execution after a given state or event until another state or event, or throughout the rest of the program if there is no subsequent occurrence of that state or event

Notice that the scope after-until describes what is often known as a weak until; it is not necessary that the state or event mentioned in the until actually happens.

In the tradition of design patterns, a specification pattern is given in the form of a package comprising its name, a precise statement of its intent, mappings into common specification formalisms, *e.g.* CTL [74] and LTL [60], examples of known uses, and relationships to other

Intent To describe a portion of a system's execution that contains an instance of certain events or states. Also known as Eventually.			
LTL Mapping	p becomes true:	Globally	$\Diamond p$
		Before r	$\neg r \,\mathsf{W}\, (p \wedge \neg r)$
		After q	$\Box(\neg q \vee \Diamond(q \wedge \Diamond p))$
		Between q and r	$\Box((q \wedge \neg r) \rightarrow (\neg r \,\mathsf{W}\, (p \wedge \neg r)))$
		After q until r	$\Box((q \wedge \neg r) \rightarrow (\neg r \,\mathsf{U}\, (p \wedge \neg r)))$
Examples and Known Uses The classic example of existence is specifying termination, *e.g.* all executions eventually reach a terminal state.			
Relationships This pattern is the dual of the Absence pattern. In many specification formalisms negation and explicit queries for Existence will be used to formulate an instance of the Absence pattern. We may wish to specify that a state/event occurs at most some bounded number of times. The Bounded Existence pattern handles that case.			

Figure 2.2: Existence property mapping

patterns. The Specification Pattern project's website [133] houses a large number of commonly occurring specification patterns. As an example, the Existence specification pattern from this website is presented here in Figure 2.2 with a mapping into LTL.

2.4 Temporal specifications in JML

Inspired by the Bandera specification patterns, and based on our experiences with specifying the temporal behaviour of the JavaCard API, we propose an extension of JML with temporal specifications. The syntax for our proposed temporal specifications is given in Figure 2.3. The specifications are clear and easy to understand, and are able to hide much of the technicalities of temporal logic. Note that the temporal specifications are associated with classes rather than individual methods; the intention is that the specifications can be written anywhere a class invariant can occur. In this section we present the informal intuition behind the different specification constructs before discussing their semantics in a more formal manner.

The basis of our logic are trace properties which describe a state property that is true for (a part of) a program's execution. Every temporal formula describes that part of the program's execution trace over which the trace property should hold, *e.g.* **after** or **before** a certain event has happened, or **between** two particular events. We use the meta-variables e and e_i for events, ϕ for temporal formulae, and ψ and χ for trace and state properties, respectively. We make a distinction between **until** or **unless**: until is what is often known as a strong until, in every execution the event has to happen; while the weaker unless can describe infinite behaviour in which the event may never happen. Notice also the asymmetry of **after** and **before** compared with **until** and **unless**: the first two can contain temporal formulae as subexpressions, whereas the latter only can contain trace properties. We chose to do this in order to keep the intuitive meaning of expressions clear and to disallow expressions such as (**before** $e_1\,\phi$) **unless** e_2, whose semantics is unclear in the case e_2 happens before e_1. A formula **between** $e_1\,e_2\,\psi$ de-

scribes the same behaviour as **after** e_1 (ψ **until** e_2). If no "trace delimiter" is specified, the property should hold over the complete trace. With every trace delimiter a set of events can be given as arguments, this means that one of these events has to happen. A special temporal formula is **atmost n** e which can be seen as syntactic sugar for **after** $e \ldots$ (**after** e **always** `false`). Here the ellipsis stands for $n-1$ occurrences of **after** e, this has the meaning that no state is reachable after a singular event e occurs more than n times.

For a method m the events that we distinguish are: m **called**, m **normal**, m **exceptional**, m **exceptional** (*excp*) and m **terminates**. We use *excp* to denote the type of exception thrown by m in the event m **exceptional** (*excp*). If no exception is specified then an occurrence of an event m **exceptional** has the meaning that m has thrown an exception of any type. In a "multiple event" two or more events are separated by commas. Multiple events have the meaning that at least one of the events listed occurs. For example, the event m **terminates** is equivalent to the multiple event m **normal**, m **exceptional** such that either an event m **normal** or an event m **exceptional** has occurred. Note that for our events, *e.g.* m **called**, we do not specify the object upon which method m is called. When we say an event m **called** occurs, we mean that m is called on some object of the class associated with the specification. Every event describes a state change whereas the trace properties describe those properties that have to hold in a sequence of states. Essentially we have **always** and **eventually** properties which describe that a property has to be true in every state of the sequence, or in at least one state of the sequence, respectively. In the syntax a keyword **never** is introduced: **never** χ being syntactic sugar for the temporal formula **always** $!\chi$. Note that `!` is the Java (and JML) negation.

The properties that hold for a particular state are ordinary JML properties, *i.e.* JML expressions with type boolean. In addition we include the following three state properties: m **enabled**, m **not enabled** and m **hangs**. For a method m, a state property m **enabled** is true in a particular state whenever: if m is called in that state and it terminates, it terminates normally. The state property m **not enabled** expresses the contrary: if the method m is called and it terminates, it will throw an exception; hence if m is known to terminate, this expresses the same as $!m$ **enabled**. A state property m **hangs** is true in a particular state whenever the method m is called in that state and it fails to terminate, *i.e.* it loops forever, or exits without returning or throwing an exception. We use the Java/JML logical connective `&` to combine state properties.

It is worth mentioning that the temporal formulae involving **before**, **until** and/or **eventually** all describe liveness properties, *i.e.* these formulae express that "something good must happen" during a program execution. This is opposed to all other temporal formulae which express safety properties, *i.e.* that "something bad will not happen" during a program execution.

2.4.1 Trace-based semantics

To give a semantics to our temporal extension we define for each possible execution trace whether it satisfies the formula. As mentioned previously, the keywords **between** and **atmost** can be expressed using **after**, **always** and **until**; therefore we need not consider these cases. Similarly, we do not explicitly give a semantics for **never**, as this can be expressed in terms of **always**. Since single-threaded Java programs are deterministic, we represent them as linear time structures. Branching time structures – which offer multiple alternate states at any given point in time – are more suitable for modelling non-deterministic systems [74]. Based on

⟨TempForm⟩ = **after** ⟨Events⟩ ⟨TempForm⟩
| **before** ⟨Events⟩ ⟨TempForm⟩
| ⟨TraceProp⟩ **until** ⟨Events⟩
| ⟨TraceProp⟩ **unless** ⟨Events⟩
| **between** ⟨Events⟩ ⟨Events⟩ ⟨TraceProp⟩
| **atmost** ⟨**nat**⟩ ⟨Event⟩
| ⟨TraceProp⟩

⟨TraceProp⟩ = **always** ⟨StateProp⟩
| **eventually** ⟨StateProp⟩
| **never** ⟨StateProp⟩

⟨Events⟩ = ⟨Event⟩
| ⟨Event⟩, ⟨Events⟩

⟨Event⟩ = ⟨method⟩ **called**
| ⟨method⟩ **normal**
| ⟨method⟩ **exceptional**
| ⟨method⟩ **exceptional** (⟨excp⟩)
| ⟨method⟩ **terminates**

⟨StateProp⟩ = ⟨JMLProp⟩
| ⟨method⟩ **enabled**
| ⟨method⟩ **not enabled**
| ⟨method⟩ **hangs**
| ⟨StateProp⟩ & ⟨StateProp⟩
| ! ⟨StateProp⟩

Figure 2.3: Temporal specification syntax

Emerson's terminology in [48], we assume that we have a linear time structure $M = (S, E, x, \rightarrow)$ where S is a set of states, E a set of events, x an infinite sequence of states $x(0), x(1), x(2), \ldots$ describing an execution of the underlying program, and $\rightarrow \subseteq S \times E \times S$ is a transition relation denoting whether a state can be reached from another state by a particular event. We assume that every state transition is labelled by a unique event.

Describing M such that it captures the full semantics of Java is beyond the scope of this thesis. However we make the assumption that it can be done. In this regard we look to (for example) Bandera, which models a Java program as a finite transition system [40]. Each state of the transition system is an abstraction of the state of the Java program. Transitions are represented by the execution of statements transforming this abstract state. We assume such transition systems can be mapped onto structures such as M in order to give an interpretation of temporal formulae for Java programs.

We define $M, x \models \phi$, meaning that the temporal formula ϕ is true of the execution trace (or path) x. When M is understood, we simply write $x \models \phi$. We use s_i to denote $x(i)$, the i^{th} element of the trace x. The notation x^j denotes the suffix path $s_j, s_{j+1}, s_{j+2}, \ldots$ and x_i^j denotes the segment path $s_i, s_{i+1}, s_{i+2}, \ldots, s_{j-1}, s_j, s_j, s_j, \ldots$ The segment path is infinite in length; it it defined such that the last state of the segment "stutters", *i.e.* it repeats infinitely often. The stutter extension is needed for traces that would otherwise have been finite.

We say that a set of events E "holds" on a state s_i (written $s_i \models E$) if and only if $i > 0$ and there exists an event e such that $s_{i-1} \xrightarrow{e} s_i$ and $e \in E$, *i.e.* the state s_i is reached by an E-transition. The set E does not hold on a state s_i (written $s_i \not\models E$) if and only if s_{i-1} and s_i are related by an event not in E, or if $i = 0$. Hence no event can hold at s_0. Moreover, E does not hold on an execution trace x (written $x \not\models E$) if and only if for all j, $s_j \not\models E$.

We overload $\models$ for defining temporal formulae, trace properties and state properties. We now define $x \models \phi$ on the structure of a temporal formula.

Definition 1 (Semantics of temporal formulae). *Given an execution path x, a set of events E, a temporal formula ϕ and a trace property ψ, we define*

$$
\begin{array}{lll}
x \models \textbf{after}\, E\, \phi & \text{iff} & \forall j.\ s_j \models E \Rightarrow x^j \models \phi \\
x \models \textbf{before}\, E\, \phi & \text{iff} & \forall j.\ s_j \models E \Rightarrow x_0^{j-1} \models \phi \\
x \models \psi\, \textbf{until}\, E & \text{iff} & \exists j.\ s_j \models E \wedge x_0^{j-1} \models \psi \\
x \models \psi\, \textbf{unless}\, E & \text{iff} & (x \models \psi \wedge x \not\models E) \vee (\exists j.\ s_j \models E \wedge x_0^{j-1} \models \psi)
\end{array}
$$

A temporal formula **after** $E\, \phi$ holds exactly in those execution traces for which – if an E-transition occurs – the temporal formula ϕ holds in the suffix path after this transition. A temporal formula **before** $E\, \phi$ holds exactly in those execution traces for which – if an E-transition occurs – the temporal formula ϕ holds in the segment path prior to this transition. A formula ψ **until** E is true for all execution traces in which an E-transition occurs and the trace property ψ is satisfied on the segment path to this E-transition. In addition, ψ **unless** E is also true if no event in E occurs and the trace property ψ holds throughout.

Definition 2 (Semantics of trace properties). *Given an execution path x and a state property χ, we define*

$$
\begin{array}{lll}
x \models \textbf{always}\, \chi & \text{iff} & \forall j.\ x^j \models \chi \\
x \models \textbf{eventually}\, \chi & \text{iff} & \exists j.\ x^j \models \chi
\end{array}
$$

The semantics for conjunction and disjunction of trace properties is standard.

This means that a state property χ is always true if it holds for all suffix paths and it is eventually true if there exists a suffix path for which it is true.

Lastly we define when a state property is satisfied on a path. We assume we have a semantics for JML (and Java) expressions and statements, described as a function $[\![-]\!]_{\mathsf{JML}}$, which defines how to evaluate an expression or statement in a particular state. In particular $[\![m]\!]_{\mathsf{JML}}$ defines the meaning of a method call. Further we assume that we have predicates: term?, signifying that a method terminates; norm?, signifying that a method terminates normally; excp?, signifying a method terminates exceptionally; and hang?, signifying that a method does not terminate at all. See [69] for an example of a semantics which allows this.

Definition 3 (Semantics of state properties). *Given an execution path x, a method name m, and a JML property ξ, we define*

$$
\begin{array}{lll}
x \models \xi & \text{iff} & [\![\xi]\!]_{\mathsf{JML}}(s_0) \\
x \models m \textbf{ enabled} & \text{iff} & \mathsf{term?}\,[\![m]\!]_{\mathsf{JML}}(s_0) \Rightarrow \mathsf{norm?}\,[\![m]\!]_{\mathsf{JML}}(s_0) \\
x \models m \textbf{ not enabled} & \text{iff} & \mathsf{term?}\,[\![m]\!]_{\mathsf{JML}}(s_0) \Rightarrow \mathsf{excp?}\,[\![m]\!]_{\mathsf{JML}}(s_0) \\
x \models m \textbf{ hangs} & \text{iff} & \mathsf{hang?}\,[\![m]\!]_{\mathsf{JML}}(s_0)
\end{array}
$$

The semantics for conjunction, disjunction and negation of state properties is standard.

Thus state predicates are evaluated in the first state of x.

2.5 Temporal aspects of the JavaCard API

The temporal extension of JML, as discussed previously, is based on our experiences writing specifications for the temporal aspects of the JavaCard API [88]. Temporal aspects are exclusively found in the JavaCard classes `JCSystem`, `Applet`, `APDU` and `OwnerPIN`. We discuss each class in detail and present their intended temporal specifications.

`JCSystem` The JavaCard `JCSystem` class includes methods which control applet execution, resource and atomic transaction management, and inter-applet object sharing on a Java smartcard [89]. Temporal aspects of the API for this class relate solely to transaction management which is of crucial importance to the card. The transaction mechanism works as follows: when an application creates or updates data on a Java smartcard, the integrity of the data needs to be preserved throughout the communication. Either all the data is updated during the communication, or in the case of an interruption, it reverts back to its initial state. A call to the method `JCSystem.beginTransaction` initiates the beginning of a set of updates. Each object update after this point is only conditionally updated: the field may appear to be updated, but the update is not yet committed. Conditionally updated fields are stored in the commit buffer. Data is committed to persistent storage and cleared from the buffer once the applet has called `JCSystem.commitTransaction`. In the case of power loss or some other system failure before this method is called, conditionally updated fields revert back to their original values. The method `JCSystem.abortTransaction` is called if the applet encounters any internal problems, *i.e.* an exception is thrown, or if it decides to cancel the transaction.

Quoted directly from the API documentation [88], the following specifications describe temporal aspects of the `JCSystem` class.

- `beginTransaction` throws `TransactionException.IN_PROGRESS` if a transaction is already in progress;
- `abortTransaction` throws `TransactionException.NOT_IN_PROGRESS` if a transaction is not in progress; and
- `commitTransaction` throws `TransactionException.NOT_IN_PROGRESS` if a transaction is not in progress.

Firstly, `beginTransaction` throws an exception if a transaction is already in progress. A transaction in progress describes that state in which `beginTransaction` has been successfully called and neither `abortTransaction` nor `commitTransaction` has yet been invoked. Hence after `beginTransaction` is called, if it is called again before `abortTransaction` or `commitTransaction`, it will throw an exception. We can write this in our proposed specification language as

after `beginTransaction` **called**
(**always** `beginTransaction` **not enabled**
unless `abortTransaction` **called**, `commitTransaction` **called**)

The above API specifications also suggest complementary behaviour: if the method `beginTransaction` is called when a transaction is not in progress, it will have normal behaviour. A transaction not in progress describes the state in which either `abortTransaction` or `commitTransaction` has been successfully called and `beginTransaction` not yet invoked. (Of course, only `beginTransaction` will be enabled initially.) In our proposed specification language we can write this complement as

after `abortTransaction` **called**, `commitTransaction` **called**
(**always** `beginTransaction` **enabled**
unless `beginTransaction` **called**)

In a similar way, based on the documentation above, specifications can be given which describe when `commitTransaction` and `abortTransaction` will, or will not be enabled. Combining these specifications with the specification for `beginTransaction` results in the following two class specifications.

```
after beginTransaction called
    (always (beginTransaction not enabled &
             abortTransaction enabled &
             commitTransaction enabled)
        unless abortTransaction called, commitTransaction called)

after abortTransaction called, commitTransaction called
    (always (beginTransaction enabled &
             abortTransaction not enabled &
             commitTransaction not enabled)
        unless beginTransaction called)
```

A JML specification for the transaction methods of the `JCSystem` class has already been given as part of the LOOP project and is outlined in [90]. This specification relies on the model field `_transactionDepth` which is assigned the values 0 and 1. When `_transactionDepth` is 0, a transaction is said to be not in progress; when it is 1, a transaction is in progress. For example, specifications for the `abortTransaction` and `commitTransaction` methods both contain the clause `//@ ensures _transactionDepth == 0;` whereas the specification for `beginTransaction` includes the clause `//@ requires _transactionDepth == 0;`. If there were more than two transaction depths, it could make for a very long and complicated specification. We believe that by hiding this beneath our temporal extension of JML makes specifications such as those for the transaction mechanism much simpler and more intuitive.

`Applet` The JavaCard `Applet` class defines an applet in JavaCard: it provides a blueprint of the variables and methods of an applet [32]. For an applet to run on a smartcard an instance of the applet needs to be first created and initialised by invoking the `Applet.install` method. This method is similar to the `main` method in a Java application. From within this method the applet is then registered with the JavaCard Runtime Environment, or JCRE, *via* the `Applet.register` method. (The JCRE encompasses the JavaCard system components that run inside a smartcard; effectively, it acts as the smartcard's operating system.) Each applet instance is identified by a unique application identifier, or AID. Once an applet is initialised and registered it can be selected and run.

Quoting directly from the API [88], the following five specifications describe temporal aspects of the `Applet` class.

- `install` must call `register`
- if `install` throws an exception before `register` is called then installation is unsuccessful
- installation is successful if call to `register` completes without an exception
- if `register` throws an exception, then installation is unsuccessful

- `register` throws `SystemException.ILLEGAL_AID` if applet instance has already called this method

Hence after the `install` method has been invoked, eventually `register` must be called by a given applet instance. Note that it is not just that we want a call to `install` to happen before a call to `register`, but we want a call to `install` to happen before a particular instance calls `register`. However in our semantics, two instances i and j calling `register` would both be considered the same event. It would be a useful extension to our specification language to be able to differentiate between the two; this is the subject of future work. We introduce the ghost (instance) field `register_called` in order to write our specification. The field represents an abstraction of the event `register` **called**. We assume: (1) that the ghost field, once it is declared, is initialised `false` (using an `initially` clause) and (2) that the first line of the `register` method is a JML `set` statement which sets `register_called` the value `true`. Thus the specification is written

after `install` **called** (**eventually** `register_called`)

Next, if `install` throws an exception before `register` is called, then the installation is unsuccessful. We use the ghost field `install_success` to represent the "success" of the installation in an abstract way. We write this specification as

after `install` **exceptional**
(**before** `register` **called** (**always** `!install_success`))

The third temporal specification says that the installation is successful within an applet instance if the call to `register` completes without an exception, *i.e.* once `register` terminates normally. We quote from the API a note given in the method detail for `install`: "Exceptions thrown by this method after successful installation are caught by the JCRE and processed by the Installer". This suggests that the installation may be deemed successful before the `install` method terminates. Hence we write this specification as

after `register` **normal** (**always** `install_success`)

The API also states that "Successful installation makes the applet instance capable of being selected via a SELECT APDU command". This does not quite match with the statement in [32] that "On successful return from the `install` method, the applet is ready to be selected and to process the upcoming APDU commands" which suggests that the applet can be selected only after `install` has returned normally. Hence we would then write our specification as

after `register` **normal**
(**after** `install` **normal** (**always** `install_success`))

However, since the API is the definitive specification, we are inclined to say that this specification is not accurate. We mention it here in order to highlight the sometimes ambiguous nature of the JavaCard specification. Our fourth specification says if `register` throws an exception,

then the installation is unsuccessful. We write this as

after `register` **exceptional** (**always** `!install_success`)

The final temporal API specification for this class states that `register` will throw an exception if the applet instance has called `register` previously. Hence the `register` method may only return normally once (if at all). Beyond this, all subsequent calls will result in an exception. We can write this as

atmost 1 `register` **normal**

It should be stressed that our intended specifications are only given for a particular applet instance. It is beyond the scope of this work to specify temporal properties of *all* created instances of a class. This problem has been tackled elsewhere however. In [41] the Bandera team introduces the notion of class instance quantification. This allows temporal specifications, written in the Bandera Specification Language, BSL, to be quantified over all instances of a particular class. The Bandera tool first adapts the model representing the Java program so that quantified variables are bound to instances of the classes named in the quantification. It then augments the temporal property ϕ to be checked by embedding it in a second temporal formula. This second temporal formula ensures that ϕ will be checked only after the quantified variables have been bound to the instances.

`APDU` An exchange of application protocol data units, or APDUs, is the means by which smartcards communicate with other computers. Methods which control communication between a smartcard and an external smartcard reader (called a host) are described by the `APDU` class [32]. For security reasons applets do not directly communicate with the applications on the host side, they instead communicate *via* the JCRE. The JCRE creates an instance of the APDU class called an APDU buffer. This buffer incorporates APDU command and response messages in an internal byte array. When receiving an APDU command from a host computer, the JCRE first writes the APDU header in the buffer. From this header, the JCRE determines whether the command is well formatted and if it can be executed. The header also conveys to the JCRE whether there is incoming command data and whether outgoing data is expected in the applet's response. If the header is "approved" the JCRE invokes the `APDU.process` method of the applet and delivers the APDU object to the applet as a method parameter. Inside the `APDU.process` method, if the incoming APDU has data, the applet can call `APDU.setIncomingAndReceive` to receive it. If the incoming APDU has more data than what can fit into the buffer, the `APDU.setIncomingAndReceive` method is followed by one or more calls to the `APDU.receiveBytes` method which manages the overflow. Since a host "times out" an applet's response if it's forced to wait too long, an applet might wish to request additional processing time from the host. It does this by calling the method `APDU.waitExtension`. An applet can call this method any time whilst processing an APDU command.

An applet can only return data to the host after it has performed the instructions specified in the command data. It calls the method `APDU.setOutgoing` to notify the host that it wants to send response data. If block chaining is not allowed (this is a method whereby the

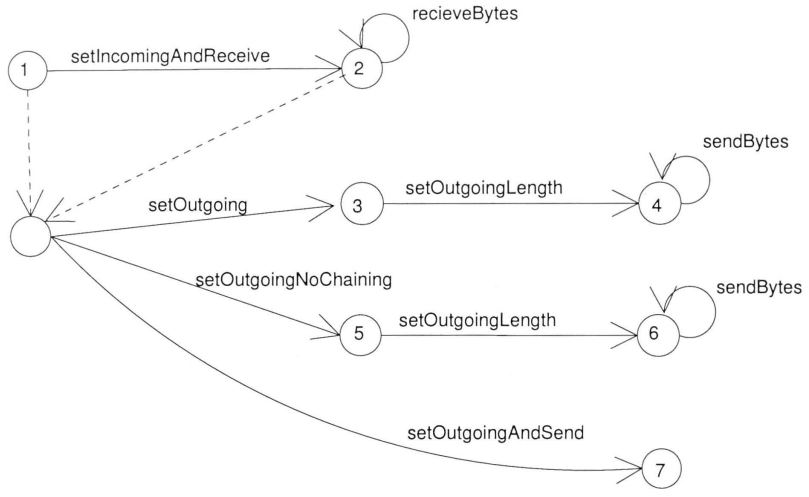

Figure 2.4: The APDU protocol

JCRE divides data into blocks and sends them *via* a chaining mechanism), then the method `APDU.setOutgoingNoChaining` should be called instead. After either of these methods are called any remaining incoming data is ignored. Following this, the applet calls `APDU.setOutgoingLength` to indicate how many response data bytes it will actually send. This also prompts the host for a GET RESPONSE command, determining whether the host is ready to receive the outgoing data. Next, the applet calls either `APDU.sendBytes` or `APDU.sendBytesLong` which actually sends out the response data. Both methods send data from the APDU buffer, but `APDU.sendBytesLong` first copies data into the APDU buffer from data stored in an applet's local buffer, or in a file. Alternatively the method `APDU.setOutgoingAndSend` may be used. This is a "shortcut" method that combines `APDU.setOutgoing`, `APDU.setOutgoingLength` and `APDU.sendBytes` into one call.

The constraints imposed upon the methods of the `APDU` class can be illustrated in the finite state machine shown in Figure 2.4. This diagram is taken from [118]. The transitions represented by dotted lines can be taken without invoking a method, *i.e.* `setOutgoing`, `setOutgoingNoChaining` and `setOutgoingAndSend` can all be invoked in state 1 or 2. Note also that the method `sendBytesLong` is not included, since its specification is similar to that of `sendBytes`. In total, there are twenty API specifications describing temporal aspects of the `APDU` class. Instead of listing them all at once, we will group them according to the type of aspects they specify. We will then discuss our proposed specifications for each group. Consider first the following five API specifications:

i. `setOutgoing` throws `APDUException.ILLEGAL_USE` if this method already invoked

ii. `setOutgoingNoChaining` throws `APDUException.ILLEGAL_USE` if this method already invoked

iii. `setOutgoingLength` throws `APDUException.ILLEGAL_USE` if this method already invoked

iv. `setIncomingAndReceive` throws `APDUException.ILLEGAL_USE` if this method already invoked

v. `setOutgoingAndSend` throws `APDUException.ILLEGAL_USE` if this method already invoked

These stipulate that `setOutgoing`, `setOutgoingNoChaining`, `setOutgoingLength`, `setIncomingAndReceive` and `setOutgoingAndSend` may only return normally once. We write these five specifications as

i. **atmost 1** `setOutgoing` **normal**
ii. **atmost 1** `setOutgoingNoChaining` **normal**
iii. **atmost 1** `setOutgoingLength` **normal**
iv. **atmost 1** `setIncomingAndReceive` **normal**
v. **atmost 1** `setOutgoingAndSend` **normal**

Recall that `setOutgoingAndSend` comprises `setOutgoing`. The specification vi. below tells us that `setOutgoingAndSend` will throw an exception if `setOutgoing` has been previously invoked. Specifications vii. and viii. tell us that either the methods `setOutgoing` or `setOutgoingNoChaining` may be called, but not both.

vi. `setOutgoingAndSend` throws `APDUException.ILLEGAL_USE` if `setOutgoing` previously invoked

vii. `setOutgoing` throws `APDUException.ILLEGAL_USE` if `setOutgoingNoChaining` already called

viii. `setOutgoingNoChaining` throws `APDUException.ILLEGAL_USE` if `setOutgoing` already invoked

The behaviour of `setOutgoingAndSend` dictates that unless `setOutgoing` has been previously called, then `setOutgoingAndSend`, if it is called, will have normal behaviour. Similarly, unless `setOutgoingNoChaining` or `setOutgoing` has been called previously, then respectively, `setOutgoing` and `setOutgoingNoChaining` will be enabled. We write these three specifications as

vi. **always** `setOutgoingAndSend` **enabled unless** `setOutgoing` **called**
vii. **always** `setOutgoing` **enabled unless** `setOutgoingNoChaining` **called**
viii. **always** `setOutgoingNoChaining` **enabled unless** `setOutgoing` **called**

Methods `setOutgoing`, `setOutgoingLength`, `sendBytes` and `sendBytesLong` must be invoked in the correct order otherwise an exception is thrown. We can see this in the following three API specifications:

ix. `setOutgoingLength` throws `APDUException.ILLEGAL_USE` if `setOutgoing` is not called

x. `sendBytes` throws `APDUException.ILLEGAL_USE` if `setOutgoingLength` is not called

xi. `sendBytesLong` throws `APDUException.ILLEGAL_USE` if `setOutgoingLength` is not called

As long as the methods are called in the right sequence they will be enabled. We write these specifications as

ix. **after** `setOutgoing` **called** (**always** `setOutgoingLength` **enabled**)
x. **after** `setOutgoingLength` **called** (**always** `sendBytes` **enabled**)
xi. **after** `setOutgoingLength` **called** (**always** `sendBytesLong` **enabled**)

Specification xii. below says that an applet instance will not receive data unless the method `setIncomingAndReceive` is called. Specification xiii. tells us that if the incoming data overflows the buffer space then `receiveBytes` will throw an exception. Specifications xiv-xvi. tell us that the applet instance will neither receive data, nor be given more time to process data, after it has indicated to the host that it wants to send data.

xii. `receiveBytes` throws `APDUException.ILLEGAL_USE` if `setIncomingAndReceive` is not called

xiii. `receiveBytes` throws `APDUException.BUFFER_BOUNDS` if not enough buffer space for incoming block size

xiv. `receiveBytes` throws `APDUException.ILLEGAL_USE` if either `setOutgoing` or `setOutgoingNoChaining` previously invoked

xv. `setIncomingAndReceive` throws `APDUException.ILLEGAL_USE` if `setOutgoing` or `setOutgoingNoChaining` previously invoked

xvi. `waitExtension` throws `APDUException.ILLEGAL_USE` if `setOutgoingNoChaining` previously invoked

In order to write specification xiii. we introduce the ghost field `buffer_space` which represents the size of the buffer space in an abstract way. Furthermore we introduce the ghost field `receiveBytes_exceptional_BUFFER_BOUNDS` which represents an abstraction of the event `receiveBytes` **exceptional** (`APDUException.BUFFER_BOUNDS`). Except for xiii. we have seen specifications of the above format before. We write them as

xii. **after** `setIncomingAndReceive` **called** (**always** `receiveBytes` **enabled**)
xiii. **always** (`!buffer_space ==> receiveBytes_exceptional_BUFFER_BOUNDS`)
xiv. **always** `receiveBytes` **enabled**
unless (`setOutgoing` **called**, `setOutgoingNoChaining` **called**)
xv. **always** `setIncomingAndReceive` **enabled**
unless (`setOutgoing` **called**, `setOutgoingNoChaining` **called**)
xvi. **always** `waitExtension` **enabled unless** `setOutgoingNoChaining` **called**

The next two specifications tell us that if a host is not ready to receive data, *i.e.* it does not return a GET RESPONSE command when prompted, methods `sendBytes` and `sendBytes Long` – when called – will throw an exception.

xvii. `sendBytes` throws `APDUException.ILLEGAL_USE` if `APDUException.NO_TO_GETRESPONSE` previously thrown

xviii. `sendBytesLong` throws `APDUException.ILLEGAL_USE` if `APDUException.NO_TO_GETRESPONSE` previously thrown

The methods `sendBytes` and `sendBytesLong` are the only two methods allowed to throw an `APDUException.NO_TO_GETRESPONSE` exception. Hence unless this exception is thrown by either of these methods, then respectively, `sendBytes` and `sendBytesLong` will be enabled. We write our specifications as

xvii. **always** `sendBytes` **enabled**
(**unless** `sendBytes` **exceptional** (`APDUException.NO_TO_GETRESPONSE`),
`sendBytesLong` **exceptional** (`APDUException.NO_TO_GETRESPONSE`))

xviii. **always** `sendBytesLong` **enabled**
(**unless** `sendBytes` **exceptional** (`APDUException.NO_TO_GETRESPONSE`),
`sendBytesLong` **exceptional** (`APDUException.NO_TO_GETRESPONSE`))

The final two API specifications tell us that `sendBytes` and `sendBytesLong` will also throw an exception if `setOutgoingAndSend` has already been called, since this method incorporates one or the other.

xix. `sendBytes` throws `APDUException.ILLEGAL_USE` if `setOutgoingAndSend` previously invoked

xx. `sendBytesLong` throws `APDUException.ILLEGAL_USE` if `setOutgoingAndSend` previously invoked

We write them in our proposed specification language as

xix. **always** `sendBytes` **enabled unless** `setOutgoingAndSend` **called**
xx. **always** `sendBytesLong` **enabled unless** `setOutgoingAndSend` **called**

`OwnerPIN` The JavaCard `PIN` interface represents a Personal Identification Number, or PIN. Any implementation must uphold a number of internal values: the PIN value; the maximum number an incorrect PIN can be presented before the PIN is blocked, otherwise known as the "try value"; the maximum PIN size; the try counter, which counts off the remaining number of times an incorrect PIN can be presented before the PIN is blocked; and the validated flag, which is true if the correct PIN is presented. The `OwnerPIN` class implements the `PIN` interface. The `OwnerPIN.check` method compares the PIN against the PIN value and returns `true` if they match. If the PIN is not blocked `OwnerPIN.check` also sets the validated flag and resets the try counter to its maximum. If it does not match `OwnerPIN.check` returns `false` and decrements the try counter, blocking the PIN when it reaches zero. The method `OwnerPIN.reset` resets the validated flag, but does nothing if the flag has not been set in the first place. The method `OwnerPIN.isValidated` returns `true` if a valid PIN has been presented since the last card reset or the last time `OwnerPIN.reset` has been called.

The single temporal API specification for the `OwnerPIN` class is as follows:

- `isValidated` returns `true` if a valid PIN has been presented since the last call to `reset`

Note that all other API specifications for this class can be written as ordinary JML specifications. We introduce the ghost fields `check_true` and `isValidated_true` initialised `false`. We presuppose the existence of a JML specification which has the precondition representing "PIN matches" and normal postcondition `check_true`. Moreover we use the JML implication `==>` in order to write this specification as

after `reset` **called** (**always** `check_true ==> isValidated_true`)

2.6 Translating temporal formulae back to JML

As mentioned before, a subset of the temporal formulae in our language extension can be translated back into standard JML. These are the temporal formulae which are used to describe safety properties, *i.e.* the formulae used to express that "something bad will not happen" during a program execution. Formulae used to describe liveness properties, *i.e.* that "something good must happen" during a program execution cannot be translated back into standard JML. Hence we do not translate temporal formulae involving **before** or **until**, nor formulae featuring **eventually**. In this section we discuss how our translation is made and as an example, show the resulting proof obligations for the temporal specification of the transaction mechanism. Our translation may seem verbose, but since it can be automatically performed by a tool, we believe this is of no importance.

In order to translate our specifications back into JML, a number of ghost fields such as `m_called`, `m_normal` and `m_exceptional_`x_i are first declared for each relevant method `m`. (A field `m_exceptional_`x_i is declared for each type of exception x_i the method `m` possibly throws.) The fields are static for static methods, and instance fields for instance methods. The fields are given a boolean value, initially `false`, and are used to trace the behaviour of the program. In the first line of every method `m` the JML set statement `//@ set m_called == true;` is added, setting `m_called`. Moreover the method `m` contains the following heavyweight behaviour specification.

```
//@ ensures m_normal&!m_exceptional_x1&...&!m_exceptional_xk;
//@ signals(x1) m_exceptional_x1&!m_normal
        ⋮
//@ signals(xk) m_exceptional_xk&!m_normal
```

The implicit default `requires` and `diverges` clauses, with respective predicates `true` and `false`, ensure that the method can always be called and when it terminates normally or abruptly, the relevant ghost fields are appropriately set.

Recall that the events in our JML temporal extension language are: `m` **called**, `m` **normal**, `m` **exceptional** (x_i), `m` **exceptional** and `m` **terminates**. If a method `m` is capable of throwing any one of the k exceptions $x_1, x_2, \ldots, x_k$ then the event `m` **exceptional** is an abbreviation for the multiple event `m` **exceptional** (x_1), `m` **exceptional** (x_2) ,..., `m` **exceptional** (x_k) where at

least one of the events listed occurs. Similarly, the event m **terminates** is an abbreviation for m **normal**, m **exceptional** (x$_1$), m **exceptional** (x$_1$) ,..., m **exceptional** (x$_k$). Recall that an event e is singular if it cannot be written as a multiple event. We also define a useful function β which assigns to each singular event e a corresponding ghost field.

$$\begin{aligned} \beta(\texttt{m}\ \textbf{called}) &= \texttt{m\_called} \\ \beta(\texttt{m}\ \textbf{normal}) &= \texttt{m\_normal} \\ \beta(\texttt{m}\ \textbf{exceptional}\ (\texttt{x}_\texttt{i})) &= \texttt{m\_exceptional\_x}_\texttt{i} \end{aligned}$$

This function will appear in the sequel.

Now let ϕ be a formula in our temporal language which expresses a safety property. Therefore we consider ϕ to be one of the following formulae: ψ, **atmost n** e, ψ **unless** e or **after** $e_1\,\phi_1$. Here ψ is a trace property, $\mathbf{n} \geq 1$ is a natural number, e and e_1 are (possibly multiple) events (singular in the case of **atmost n** e), and ϕ_1 is some temporal formula describing a safety property. Note that we are unable to translate the formula **between** $e_1\,e_2\,\psi$ into standard JML because we cannot guarantee the second event will happen. Recall that **between** $e_1\,e_2\,\psi$ describes the same behaviour as **after** e_1 (ψ **until** e_2). We can however translate **after** e_1 (ψ **unless** e_2) where the second event e_2 may or may not occur.

We first define the general translation for a given formula ϕ, then examine the translation of each of the formulae ψ, **atmost n** e, ψ **unless** e and **after** $e_1\,\phi_1$ in detail. Let C be the function which returns the JML clause for a given formula ϕ and let the function α return the content of that clause. We define ϕ's translation into standard JML as

$$\begin{aligned} \phi_{JML} \quad = \quad & \texttt{//@}\ C(\phi)\ \alpha(\phi); \\ & \mathcal{A}(\phi) \end{aligned}$$

The main "//@ $C(\phi)$ $\alpha(\phi)$;" part of the translation is either a behaviour method specification or a class specification, *i.e.* an invariant. We choose to translate into behaviour specifications since it is possible to desugar all other method specification cases into behaviour specifications [122]. Note that the implicit clauses of all behaviour specifications retain their default values. Additional specifications are listed in $\mathcal{A}(\phi)$. These are either incorporated into the method specification itself, or are class specifications. Usually additional specifications declare or set the model variables that appear in the main translation. In the case of the translation of **always** m **hangs**, the additional specifications are used to override the default values of a number of implicit clauses.

Translating ψ Since we are only translating the temporal formulae which describe safety properties back into standard JML, and since a trace property **never** χ in our language is simply just syntactic sugar for **always** ! χ where ! is the Java and JML negation, the only trace property ψ that we need to consider here is **always** ($\chi_1 \& \chi_2 \ldots \& \chi_t$) where χ_i for all $i = 1, 2, \ldots, t$ is one of the following state properties: m **enabled**; m **not enabled**; m **hangs**; or ξ, an ordinary JML property, *i.e.* a JML expression of type boolean. (Recall that we use the Java and JML logical connective & to combine state properties.) We suppose that the method m is allowed to throw any of the k possible exceptions $\texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_\texttt{k}$ where $k \geq 0$. We define the following

translation.

$$
\begin{aligned}
(\textbf{always}\ (\chi_1 \& \chi_2 \& \ldots \& \chi_t))_{JML} &= (\textbf{always}\,\chi_1)_{JML} \\
&\quad\ (\textbf{always}\,\chi_2)_{JML} \\
&\quad\ \vdots \\
&\quad\ (\textbf{always}\,\chi_t)_{JML} \\
(\textbf{always}\,\chi_i)_{JML} &= \texttt{//@}\ C(\textbf{always}\,\chi_i)\ \alpha(\textbf{always}\,\chi_i); \\
&\quad\ \mathcal{A}(\textbf{always}\,\chi_i)
\end{aligned}
$$

We define the mappings of C and α for each trace property as follows. We also define the additional specifications for each trace property. By $\mathcal{A}(\psi) = \emptyset$ we mean that the trace property ψ has no additional specifications.

$$
\begin{aligned}
C(\textbf{always}\,\xi) &= \texttt{invariant} \\
\alpha(\textbf{always}\,\xi) &= \xi \\
C(\textbf{always}\,\texttt{m}\ \textbf{enabled}) &= \texttt{signals(}x_1\texttt{)} \\
&\quad\ \vdots \\
&\quad\ \texttt{signals(}x_k\texttt{)} \\
\alpha(\textbf{always}\,\texttt{m}\ \textbf{enabled}) &= \texttt{false} \\
&\quad\ \vdots \\
&\quad\ \texttt{false} \\
C(\textbf{always}\,\texttt{m}\ \textbf{not enabled}) &= \texttt{ensures} \\
\alpha(\textbf{always}\,\texttt{m}\ \textbf{not enabled}) &= \texttt{false} \\
C(\textbf{always}\,\texttt{m}\ \textbf{hangs}) &= \texttt{diverges} \\
\alpha(\textbf{always}\,\texttt{m}\ \textbf{hangs}) &= \texttt{true} \\
\mathcal{A}(\psi) &= \text{if } \psi = \textbf{always}\,\texttt{m}\ \textbf{hangs} \\
&\quad\ \text{then } \texttt{//@ ensures false;} \\
&\qquad\ \texttt{//@ signals(}x_1\texttt{) false;} \\
&\qquad\ \vdots \\
&\qquad\ \texttt{//@ signals(}x_k\texttt{) false;} \\
&\quad\ \text{else } \emptyset
\end{aligned}
$$

Also we define $(\textbf{always}\,\xi)_{JML}$ as a class specification and $(\textbf{always}\,\texttt{m}\ \textbf{enabled})_{JML}$, $(\textbf{always}\,\texttt{m}\ \textbf{not enabled})_{JML}$ and $(\textbf{always}\,\texttt{m}\ \textbf{hangs})_{JML}$ as method `m` specifications. Hence the trace property $\textbf{always}\,\xi$ translates into a class invariant whereby the JML property ξ is true in each state outside of a public method's execution. Note that this translation is not entirely faithful due to the semantics of an invariant. It is possible for an invariant to break within a method's execution, however invariants must hold at: the end of a constructor; the beginning and end of methods; and at the point of a method call within methods and constructors. By translating $\textbf{always}\,\xi$ into an invariant we can verify that ξ holds at these points. The semantics of $\textbf{always}\,\xi$ is weakened, but verification of this temporal property would be impossible otherwise.

The trace property $\textbf{always}\,\texttt{m}\ \textbf{enabled}$ translates into a behaviour method specification such that all k `signals` clauses are `false`. Hence if the method is called, it will not throw an ex-

ception of any type. A trace property **always** `m` **not enabled** translates into a behaviour method specification with `ensures` clause `false`. Hence if `m` is called, it will not terminate normally. A trace property **always** `m` **hangs** translates into a behaviour method specification such that its `diverges` clause is `true`. The default values of the implicit `ensures` and `signals` clauses (both `true`) are overridden by the additional clauses for the formula **always** `m` **hangs** which stipulate that the method will neither terminate normally nor abnormally. Hence if `m` is called, it will hang. These translations coincide with our understanding of a method being enabled, not enabled, or hung.

Translating atmost n *e* We next consider ϕ as the expression **atmost n** *e* where *e* is a singular event. We define the translation of **atmost n** *e* below. Recall that β assigns to each singular event *e* a corresponding ghost field.

$$
\begin{array}{rcl}
(\textbf{atmost n}\, e)_{JML} & = & \texttt{//@}\ C(\textbf{atmost n}\, e)\ \alpha(\textbf{atmost n}\, e); \\
 & & \mathcal{A}(\textbf{atmost n}\, e) \\
 & & \\
C(\textbf{atmost n}\, e) & = & \texttt{ensures} \\
\alpha(\textbf{atmost n}\, e) & = & \texttt{times\_}\beta(e)\texttt{>n==>false} \\
\mathcal{A}(\textbf{atmost n}\, e) & = & \texttt{//@ ghost int times\_}\beta(e)\texttt{;} \\
 & & \texttt{//@ initially times\_}\beta(e)\texttt{==0;} \\
 & & \texttt{//@ constraint} \\
 & & \qquad \texttt{times\_}\beta(e)\texttt{==\old(times\_}\beta(e)\texttt{)+1} \\
 & & \qquad [\texttt{for } \textit{method\_e}];
\end{array}
$$

The additional specifications are class specifications. They declare the integer ghost field `times_`$\beta(e)$, initialise it at zero and place a constraint upon it so that it may only increment by 1 each time the event occurs. The constraint is such that it is only respected by the method *method_e*, which is the method cited in the event *e*. The "`//@` C(**atmost n** *e*) α(**atmost n** *e*);" part of the translation ensures that if `times_`$\beta(e)$ is greater than *n*, *i.e.* the event *e* has occurred more than *n* times, then the method will not terminate normally. This specification is a behaviour method specification corresponding to the method cited in *e*.

For example, a specification **atmost 3** (`m` **called**) will translate into two JML specifications. The first of these is a class specification.

```
//@ ghost int times_m_called;
//@ initially times_m_called==0;
//@ constraint times_m_called==\old(times_m_called)+1 [for m];
```

Here the ghost field `times_m_called` is declared and initialised. The constraint (which is only respected by the method `m`) is such that every time the method is called, the field `times_m_called` is incremented by 1. When this field becomes greater than 3, then the second JML specification – a method `m` behaviour specification shown below – ensures that if the method is called then it will not terminate normally.

```
//@ ensures times_m_called>3==>false;
```

Translating ψ unless e For the JML translation of the formula ψ **unless** e, we only consider ψ as the trace property **always** $(\chi_1 \& \chi_2 \& \ldots \& \chi_t)$ where χ_i for all $i = 1, 2, \ldots, t$ is one of the following state properties: `m` **enabled**; `m` **not enabled**; `m` **hangs**; or ξ, an ordinary JML property. Recalling that JML uses the Java conditional operators `&&` and `||` which evaluate their right-hand operand only if the value of the expression has not already been determined by the left-hand operand, we let e be the multiple event $e_1, e_2, \ldots, e_s$ and define

$$
\begin{aligned}
(\textbf{always}\ (\chi_1 \& \chi_2 \& \ldots \& \chi_t)\ \textbf{unless}\ e)_{JML} &= (\textbf{always}\ \chi_1\ \textbf{unless}\ e)_{JML} \\
&\quad (\textbf{always}\ \chi_2\ \textbf{unless}\ e)_{JML} \\
&\quad \vdots \\
&\quad (\textbf{always}\ \chi_t\ \textbf{unless}\ e)_{JML} \\
(\textbf{always}\ \chi_i\ \textbf{unless}\ e)_{JML} &= \texttt{//@}\ C(\textbf{always}\ \chi_i\ \textbf{unless}\ e)\ \alpha(\textbf{always}\ \chi_i\ \textbf{unless}\ e); \\
&\quad \mathcal{A}(\textbf{always}\ \chi_i\ \textbf{unless}\ e) \\
C(\textbf{always}\ \chi_i\ \textbf{unless}\ e) &= C(\textbf{always}\ \chi_i) \\
\alpha(\textbf{always}\ \chi_i\ \textbf{unless}\ e) &= \texttt{!}(\beta(e_1)\ \texttt{||}\ \beta(e_2)\ \texttt{||}\ \ldots\ \texttt{||}\ \beta(e_s))\ \texttt{=>}\ \alpha(\textbf{always}\ \chi_i) \\
\mathcal{A}(\textbf{always}\ \chi_i\ \textbf{unless}\ e) &= \emptyset
\end{aligned}
$$

Hence as long as one of the events of e have not occurred, *i.e.* the values of the ghost fields assigned to each event remain `false`, then **always** χ_i holds. The specification is placed according to the nature of χ_i. For example if χ_i is ξ then the specification is a class invariant. If χ_i is one of the following: `m` **enabled**, `m` **not enabled**, or `m` **hangs**, then the specification is a `m` behaviour specification.

As an example, a specification **always** (`m` **not enabled** `&` `n` **enabled**) **unless** (`o` **called**, `p` **exceptional** (`x`)) will translate into two JML specifications. The first of these comprises a `m` behaviour specification.

```
//@ ensures!(o_called||p_exceptional_x)==>false;
```

Hence, as long as `o_called` and `p_exceptional_x` remain `false`, *i.e.* the events they represent have not occurred, then `m` (if it is called) will not terminate normally. The second specification given below comprises a `n` behaviour specification. Supposing that the method `n` may throw exceptions x_2 and x_3, then as long as `o_called` and `p_exceptional_x` remain `false`, then `n` (if it is called) will not throw an exception of any type.

```
//@ signals(x2)!(o_called||p_exceptional_x)==>false;
//@ signals(x3)!(o_called||p_exceptional_x)==>false;
```

Translating after $e_1\ \phi_1$ Now let us consider the formula **after** $e_1\ \phi_1$ where e_1 may be a multiple event and ϕ_1 describes some safety property. Therefore ϕ_1 is one of the following: ψ, **atmost n** e_2, ψ **unless** e_2, or **after** $e_2\ \phi_2$. Here ϕ_2 describes a safety property and e_2 may also be considered a multiple event (it is singular in the case of **atmost n** e_2). Note that since we are only translating safety properties and the keyword **never** is an abbreviation for **!always**, the only trace property ψ that we consider here is **always** $(\chi_1 \& \chi_2 \& \ldots \& \chi_t)$ where χ_i for all $i = 1, 2, \ldots, t$ is one of the following state properties: `m` **enabled**, `m` **not enabled**, `m` **hangs**, or

ξ. Let e_1 be the multiple event $e_{11}, e_{12}, \ldots, e_{1s}$. For $\phi_1 \equiv$ **after** e_2 (**after** e_3 ($\ldots$ (**after** $e_z \eta$))) where z is an arbitrary natural number including zero, we call η the "innermost" formula of ϕ_1. Note that η is one of the following formula: **always** $(\chi_1 \& \chi_2 \& \ldots \& \chi_t)$, **atmost n** e_{z+1}, or **always** $(\chi_1 \& \chi_2 \& \ldots \& \chi_t)$ **unless** e_{z+1}. We first define

$$\begin{aligned}(\textbf{after}\, e_1\, \phi_1)_{JML} \quad = \quad & (\textbf{after}\, e_1\, \phi_1^1)_{JML} \\ & (\textbf{after}\, e_1\, \phi_1^2)_{JML} \\ & \vdots \\ & (\textbf{after}\, e_1\, \phi_1^t)_{JML}\end{aligned}$$

Where ϕ_1^j is equivalent to ϕ_1 for all $j = 1, 2, \ldots, t$ except that

- if **always** $(\chi_1 \& \chi_2 \& \ldots \& \chi_t)$ is the innermost formula of ϕ_1 then **always** χ_j is the innermost formula of ϕ_1^j
- if **always** $(\chi_1 \& \chi_2 \& \ldots \chi_t)$ **unless** e_{z+1} is the innermost formula of ϕ_1 then **always** χ_j **unless** e_{z+1} is the innermost formula of ϕ_1^j
- if **atmost n** e_{z+1} is the innermost formula of ϕ_1 then $t = 1$ and $\phi_1^1 = \phi_1$

For example, if ϕ_1 is the formula **after** e_2 (**always** (`m` **enabled** `&n` **not enabled**) **unless** e_3) then ϕ_1^1 is **after** e_2 (**always** `m` **enabled unless** e_3) and ϕ_1^2 is **after** e_2 (**always** `n` **not enabled unless** e_3). Also,

$$\begin{aligned}&(\textbf{after}\, e_1\ (\textbf{after}\, e_2\ (\textbf{always}\ (\texttt{m}\ \textbf{enabled}\ \&\texttt{n}\ \textbf{not enabled})\ \textbf{unless}\, e_3)))_{JML} \\ &\quad = (\textbf{after}\, e_1\ (\textbf{after}\, e_2\ (\textbf{always}\, \texttt{m}\ \textbf{enabled unless}\, e_3)))_{JML} \\ &\quad\ \ (\textbf{after}\, e_1\ (\textbf{after}\, e_2\ (\textbf{always}\, \texttt{n}\ \textbf{not enabled unless}\, e_3)))_{JML}\end{aligned}$$

The formula ϕ_1^j has the form **after** e_2 ($\ldots$ (**after** $e_z \eta^j$)) where z is an arbitrary natural number including zero and η^j is one of the following: **always** χ_j, **always** χ_j **unless** e_{z+1}, or **atmost n** e_{z+1}. Let e_2 be the multiple event $e_{21}, e_{22}, \ldots, e_{2p}$ and let e_z be the multiple event $e_{z1}, e_{z2}, \ldots, e_{zq}$. Hence we finally define for all $j = 1, 2, \ldots, t$

$$\begin{aligned}(\textbf{after}\, e_1\, \phi_1^j)_{JML} \quad &= \quad //@\ C(\textbf{after}\, e_1\, \phi_1^j)\ \alpha(\textbf{after}\, e_1\, \phi_1^j); \\ &\qquad \mathcal{A}(\textbf{after}\, e_1\, \phi_1^j) \\ C(\textbf{after}\, e_1\, \phi_1^j) \quad &= \quad C(\eta^j) \\ \alpha(\textbf{after}\, e_1\, \phi_1^j) \quad &= \quad (\backslash\texttt{old}(\beta(e_{11}))\ ||\ \backslash\texttt{old}(\beta(e_{12}))\ ||\ \ldots\ ||\ \backslash\texttt{old}(\beta(e_{1s}))) \\ &\qquad ==> (\backslash\texttt{old}(\beta(e_{21}))\ ||\ \backslash\texttt{old}(\beta(e_{22}))\ ||\ \ldots\ ||\ \backslash\texttt{old}(\beta(e_{2p}))) \\ &\qquad\quad ==> \ldots \\ &\qquad\qquad ==> (\backslash\texttt{old}(\beta(e_{z1}))\ ||\ \backslash\texttt{old}(\beta(e_{z2}))\ ||\ \ldots\ ||\ \backslash\texttt{old}(\beta(e_{zq}))) \\ &\qquad\qquad\quad ==> \alpha(\eta^j)\end{aligned}$$

Hence the formula **after** $e_1\, \phi_1^j$ is translated into a JML specification such that if one of the ghost fields representing a singular event of e_1 is `true` in the pre-state of the method, *i.e.* an event of e_1 has occurred, then ϕ_1^j holds. The type of specification and its location is dependent on the nature of η^j. If η^j is **always** ξ_j then the specification is a class invariant. If η^j is either

always χ_j or **always** χ_j **unless** e, where χ_j is one of the following: `m` **enabled**, `m` **not enabled**, or `m` **hangs**, then the specification is a method `m` behaviour specification. If η^j is **atmost n** e then the specification becomes a method behaviour specification for the method cited in the event e. We define the additional specifications as follows:

- if **always** χ_j is the innermost formula of ϕ_1^j then $\mathcal{A}(\textbf{after}\ e_1\ \phi_1^j) = \mathcal{A}(\textbf{always}\ \chi_j)$
- if **atmost n** e_{z+1} is the innermost formula of ϕ_1^j then $\mathcal{A}(\textbf{after}\ e_1\ \phi_1^j) = \mathcal{A}(\textbf{atmost n}\ e_{z+1})$
- if **always** χ_j **unless** e_{z+1} is the innermost formula of ϕ_1^j then

$$\begin{aligned}\mathcal{A}(\textbf{after}\ e_1\ \phi_1^j) \quad = \quad & \texttt{//@ set}\beta(e_{11})\texttt{==false;}\\ & \texttt{//@ set}\beta(e_{12})\texttt{==false;}\\ & \quad\vdots\\ & \texttt{//@ set}\beta(e_{1s})\texttt{==false;}\\ & \texttt{//@ set}\beta(e_{21})\texttt{==false;}\\ & \texttt{//@ set}\beta(e_{22})\texttt{==false;}\\ & \quad\vdots\\ & \texttt{//@ set}\beta(e_{2p})\texttt{==false;}\\ & \texttt{//@ set}\beta(e_{z1})\texttt{==false;}\\ & \texttt{//@ set}\beta(e_{z2})\texttt{==false;}\\ & \quad\vdots\\ & \texttt{//@ set}\beta(e_{zq})\texttt{==false;}\end{aligned}$$

We will use the specification of the transaction mechanism to illustrate our translation of **after** $e_1\ \phi_1$. Consider the temporal formula **after** e_1 (ψ **unless** e_2) where e_1 is a singular event and e_2 is the multiple event e_{21}, e_{22}. Following the translation outlined above, this becomes

```
//@ C(ψ) \old(β(e1)) ==> (!(β(e21)||β(e22)) ==> α(ψ));
//@ setβ(e1)==false;
```

The second annotation is the additional specification. The reasoning behind it is thus: in the specification **after** e_1 (ψ **unless** e_2), if one of the events of e_2 happen, then e_1 is no longer true. The additional specification ensures that at the moment one of the events of e_2 occur, e_1 is set to `false` again. Where this specification is listed is determined by the methods cited in e_2. For example, **after** `m` **called** (**always** `n` **not enabled unless** (`o` **called**, `p` **called**)) translates into the following JML.

```
//@ ensures \old(m_called) ==> (!(o_called||p_called) ==>false);
//@ set m_called==false;
```

The first annotation is a method `n` behaviour specification, whereas the set statement needs to be added in the beginning of methods `o` and `p`.

In the case of the transaction mechanism, the specifications follow a similar format. Figure 2.5 shows their translation into JML. The specification for `commitTransaction` is not included as it is similar to that for `abortTransaction`.

```
class JCSystem {

/*@ public behavior
    ensures
      \old(beginTransaction_called) ==>
        (!(abortTransaction_called||commitTransaction_called)
                                                    ==>false);
    signals(Exception)
      (\old(abortTransaction_called) ||\old(commitTransaction_called))
                              ==>(!beginTransaction_called==>false);
 */
     beginTransaction() {
     //@ set beginTransaction_called==true;
     //@ set abortTransaction_called==false;
     //@ set commitTransaction_called==false;
     ..
     }

/*@ public behavior
    ensures
      (\old(abortTransaction_called) ||\old(commitTransaction_called))
                              ==>(!beginTransaction_called==>false);

    signals(Exception)
      \old(beginTransaction_called) ==>
        (!(abortTransaction_called||commitTransaction_called)
                                                    ==>false);
 */
     abortTransaction() {
     //@ set abortTransaction_called==true;
     //@ set beginTransaction_called==false;
     ..
     }

// commitTransaction similar to abortTransaction

}
```

Figure 2.5: JML specification for the transaction mechanism

In our discussion on related work in Section 2.1 we mentioned that Bellegarde, Groslambert *et al.* had recently proposed a way to verify liveness properties expressed using our extension language [15]. Now that we have sufficient background knowledge it is worth discussing their approach in greater detail. Recall that they verify a class's liveness property by decomposing it into two tasks. The first of these is showing that the class itself is run in an "ideal" environment, *i.e.* an environment that calls all methods sufficiently often. If this is the case, then the liveness property may be established. The second task is checking whether the class itself establishes the liveness property. Task 1 is addressed by translating both concrete and ideal systems into "B" models [1] and showing that the concrete system is a refinement of the ideal system. Task 2 is addressed by generating JML annotations that are sufficient to guarantee the liveness property. Giorgetti and Groslambert's JAG tool (JML Annotation Generator) has recently been developed for this purpose [59].

JAG accepts an extension of our specification language: for each liveness property an additional invariant and variant is required. The invariant describes a property that is maintained until the liveness event is satisfied. The variant is an expression that strictly decreases with each method invocation, ensuring that eventually "something good must happen". JAG reduces each temporal property into semantically-equivalent primitives. The Inv primitive represents the safety part of the property, whereas the Loop primitive represents the liveness part. The Witness primitive acts as a past marker on the class. Each Inv primitive is translated into an invariant, and each Loop primitive is translated into a set of invariants and constraints which specify that the variant decreases. Each Witness primitive is translated into a ghost field.

Hence the formula **after** e_1 (**always** ψ **until** e_2) **under invariant** x **variant** y is verified by showing that for any execution in which e_1 happens: (1) eventually e_2 occurs; and (2) as long as e_2 does not happen, ψ remains true. The latter of these is simply the verification of the safety property **after** e_1 (**always** ψ **unless** e_2). The former requires showing that the invariant is preserved and the variant strictly decreases *via* JAG using the Jack tool, discussed in Section 2.2.1, as a back-end theorem prover.

2.7 Conclusions and future work

We have presented an extension of JML with temporal logic. The extension is based on the specification patterns proposed within the Bandera project and is tailored especially for Java. Although not all specifications in our language extension can be translated back into standard JML, we believe that by using our syntax, specifications are more readable and intuitive. Our language extension also allows the specification of liveness properties – eventually something good will happen – which currently cannot be specified in standard JML.

We have gained considerable experience in writing specifications for the JavaCard API and have provided specifications for all temporal aspects of the language. We have described a semantics for our extension language and have discussed how temporal properties can be verified if they can be translated back into standard JML.

It is future work to extend the language overall, allowing trace properties to state that events eventually or never happen, and to integrate our language extension with a runtime assertion generator, as is done for trace assertions in the Jass project [86] and in the Java PathExplorer project [66].

Chapter 3

Factorising temporal specifications

In this chapter we propose a method to factorise the verification of temporal properties for multi-threaded programs over their different threads. Essentially the method involves showing that some of the threads establish the property, while the other threads do not affect it. The method is fine-tuned by identifying – for each property – particular conditions necessary for preservation. As a specification language we again used the specification patterns developed by the Bandera team. The language differs somewhat from the JML extension language presented in Chapter 2 since that language was specifically designed with single-threaded JavaCard in mind. There we wanted to be able to express typical specifications of smartcards, *e.g.* the transaction mechanism. Here we are concerned with multi-threaded programs and our target specification language is correspondingly more generalised. For each specification pattern a decomposition rule is proposed. The soundness of each rule using the pattern mappings as defined for LTL has been shown. The proofs have been formalised using the theorem prover Isabelle/HOL. We direct the reader to a full set of theory files which can be found at `ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/Factorisation/`.

The chapter expands generally upon an existing paper written with Marieke Huisman [72]. Here we describe our program model and factorisation method in much greater detail and also – in contrast to the paper – prove a number of interesting results arising from our formalism.

3.1 Motivation

In the previous chapter we looked at making a specification language more expressive whilst still maintaining that language's verifiability. A downside to this is that what we gain by making a specification language more expressive, we often lose by making the verification task more difficult and/or complex. This is further compounded when it comes to verifying multi-threaded applications: the various tools and techniques which have been developed that allow one to formally verify realistic applications fail to scale up because the verification of a multi-threaded application faces the state explosion problem. This requires one to consider all the possible interleavings of the different threads, all running in parallel.

To make verification of multi-threaded applications feasible, a number of techniques can be be used to lighten the proof burden. First of all there are abstraction techniques which reduce the possible state space of a program; see *e.g.* [34]. Second, there are slicing techniques which remove those instructions that are irrelevant to the property being checked; see *e.g.* [63].

Thirdly, recent work on controlling thread interference proposes the use of atomicity checkers that establish whether the outcome of a method can be affected by interleavings with other threads; see [53; 65]. This last development is important because any atomic method can be verified in isolation without considering the possible interleavings.

We advocate an alternative approach: in order to simplify the verification tasks we factorise the temporal specifications for the whole system into specifications for a subset of the threads. This is done by defining rules of the form

$$\frac{\mathcal{T}_1 \models \phi \qquad C \models \mathcal{T}_2 \text{ preserves } V}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

Here ϕ is a temporal specification, $\mathcal{T}_1$ and $\mathcal{T}_2$ are sets of threads, V is a set of variables, and C is a set of additional conditions under which the set of variables V should not be changed. Such rules state that if we wish to verify whether a composed system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies a temporal property ϕ it is sufficient to show that one can decompose the system into $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\mathcal{T}_1$ satisfies ϕ, while $\mathcal{T}_2$ does not affect the validity of ϕ. The latter follows from showing that $\mathcal{T}_2$ preserves a set of variables V which depends on the property ϕ and on the threads in $\mathcal{T}_1$. It is beyond the scope of this thesis to discuss in detail how this set V can be constructed, instead it is assumed that an appropriate dependency analysis is available; see *e.g.* [63].

Consider, for example, the behaviour of a bounded channel which receives messages and delivers them whenever possible. Typically this channel would not depend on the surrounding components that create and consume the messages. The factorisation method presented here allows us to verify the channel thread in isolation, without considering interleavings with the consumer and producer threads. Moreover, if new threads are added to the application we do not have to re-do the verifications, we need only verify that the new threads do not interfere in an unwanted way, *i.e.* we have to show that under particular conditions that depend on the property, the new thread does not affect the variables related to the property. For example, suppose we are verifying the property "if the bounded channel is full, it will eventually become non-full". The factorisation method allows us to reduce this to: (1) verifying the property for the bounded channel thread, using any standard verification technique; and (2) proving that all other threads do not affect the channel *whenever* the channel is full. The possibility to put conditions on the factorisation distinguishes this method from, *e.g.* slicing. We believe that checking whether a thread does not affect certain variables can be done efficiently using techniques to check frame conditions; see *e.g.* [135] for a sound method to verify frame conditions.

The model we use to represent multi-threaded applications is inspired by the programming language Java [61]. In Java there is an arbitrary number of threads, all running in parallel and all using the same shared, global memory. Data can be protected by a lock; only one thread at a time can hold such a lock. The set of possible executions of a Java program is the set of all possible interleavings of the sequential threads; the threads are only restrained by the requirements on the locks.

The program model and the temporal specification language have been formalised in the interactive theorem prover Isabelle/HOL [109]. Moreover, the proof rules presented here have all been proven correct with respect to our formalisation.

Related work Our work is directly inspired by a compositional verification method for Unity [120; 119]. However the program model of Unity composes non-deterministic processes. Each action in a process is considered to be atomic (even when it has multiple side-effects) and can be executed repeatedly. This is in contrast to our Java model where each process is sequential and the level of atomicity is prescribed by the Java memory model [61].

We have also been influenced by Santone's compositional approach to verification of concurrent systems, specified using the selective μ-calculus [127]. However Santone's approach focuses on processes with synchronised communication, while we concentrate on the shared value model. Furthermore, we mention compositional model checking approaches; see *e.g.* [35]. These differ from our approach in that they assume an arbitrary specification for each component and then show that these local specifications are sufficient to ensure global correctness. In contrast, our approach shows under which conditions it is sufficient to verify a global property only on part of the system.

As explained previously our work differs from existing approaches to abstraction, slicing and controlling thread interference in that it does not the consider the whole application as a single unit. These techniques all aim at reducing the verification burden by eliminating unnecessary verification tasks, while our technique aims at decomposing the program into different parts for which different verification tasks exist. Because they encompass large and formidable areas of research – especially abstraction and slicing – we discuss these "non-compositional" techniques below in greater detail.

Abstraction The main goal of abstraction is to avoid the construction of a full-blown system model. An abstract interpretation function α maps a program P directly to an abstraction of the transition system representing P. The abstraction is such that if a property holds for the abstract system, then a corresponding property holds for the original system (the reverse is usually not the case). By ignoring state information irrelevant to the property being checked, the abstraction is smaller and more manageable compared to the original system.

Slicing Program slicing was originally introduced by Weiser [144]. He defined a program slice as a "reduced, executable program" obtained from a program P by removing statements such that the slice still replicates some of the behaviour of P. Slightly varying notions of program slices have since arisen, but it is now generally agreed upon that a program slice consists of those parts of a program that (potentially) affect the values computed at some point of interest, called the slicing criterion [138]. A main distinction is that of static versus dynamic program slicing: static slicing makes no assumptions regarding the program's input, whereas dynamic slicing relies on a specific test case. Weiser's original method of computing a static program slice was by identifying consecutive sets of indirectly relevant statements according to data flow and control flow dependencies. These dependencies are defined in terms of the Control Flow Graph (CFG) of a program. A CFG is comprised of nodes which represent each statement and control predicate in a program. Control predicates are assertions which explicitly mention the control state. Each CFG also contains the special nodes *start* and *stop* corresponding to the beginning and end of a program, respectively. An edge from node n to m indicates the possible flow of control from n to m.

Ottenstein and Ottenstein later restated the problem of computing a program slice in terms of reachability in a Program Dependence Graph, or PDG [114]. A standard PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies. The slicing criterion is identified as a set of nodes $\{n_1, n_2, \ldots n_k\}$ and computing a slice involves finding all nodes upon which the statement at each node n_i depend. These are found by computing the transitive closure of the dependencies in the PDG with respect to each n_i. Many program slicing approaches employ modifications or extensions of PDGs as their underlying program representation.

Thread interference Previous work on controlling thread interference has focused primarily on race conditions. Race conditions are created when two threads simultaneously access the same data variable and at least one of the accesses is a write. A number of tools have been developed to detect such conditions in Java including the Race Condition Checker [52], ESC/Java [54] and Calvin [57]. However detecting race conditions alone is not enough to guarantee the absence of errors caused by unexpected thread interleavings. The proposal by Flanagan and Qadeer [55] to address the non-interference property of atomicity attempts to remedy the situation. A method is said to be atomic when any interleaving of its instructions with other threads gives the same result as executing its instructions without interleavings. Hence reasoning about an atomic method's behaviour in a multi-threaded context is reduced to the easier task of reasoning about a method's sequential behaviour. A type system for specifying and verifying the atomicity of methods in multi-threaded Java has been devised and implemented in the dynamic atomicity checker "Atomizer" [53]. More recently, Hatcliff *et al.* have adapted the Bogor model checker (instrumental to the Bandera tool set) to detect atomicity violations [65].

Notice that the approaches we have discussed (including our own) are not mutually exclusive. In fact we advocate the combined use of the different techniques mentioned. Note however that a combination of non-compositional approaches alone will still require the consideration of the application as a single unit. Its verification will lack the flexibility our method provides.

The remainder of this chapter is organised as follows. The next section introduces the multi-threaded program model and discusses how this relates to Java. Next, Section 3.3 introduces the temporal logic that we use to specify program properties. Section 3.4 discusses the proof rules that we use to factorise temporal specifications, whereas Section 3.5 discusses the formalisation and verification of the method. Section 3.6 shows how our method works in practice. Proofs of intermediary results that we use throughout the chapter are given in Section 3.7. Finally Section 3.8 draws conclusions, shows how our method can be used in a larger context, and discusses future work.

3.2 The program model

Programs are represented by labelled transition systems (LTSs). Each thread in the program is represented by a single LTS and the program itself is represented as their composition. We briefly recall some definitions before moving on to discuss why we assume Java programs can

be represented as LTSs.

3.2.1 Labelled transition systems

Definition 4 (LTS). *A* Labelled Transition System (LTS) *is a 4-tuple* $\mathcal{T} = (S, A, \rightarrow, I)$ *where* S *is a non-empty set of states,* A *is a set of action labels,* $\rightarrow \subseteq S \times A \times S$ *is the transition relation denoting whether a state can be reached from another state by a particular action, and* $I \subseteq S$ *is the set of initial states.*

We say an action $a \in A$ *is* enabled *in state* s*, denoted* $\mathsf{enabled}\,\mathcal{T}\,s\,a$*, if there is a state* t *such that* $(s,a,t) \in \rightarrow$*. For convenience we write* $s \xrightarrow{a} t$ *for* $(s,a,t) \in \rightarrow$*.*

Inasmuch as we are modelling Java programs it is assumed: (1) we have a single global shared memory, and (2) composition of LTSs is defined only for systems with the same state space. The transition relation in the composed LTS is defined as the union of the two individual transition relations, while initial states are defined as the intersection of the individual initial states. This ensures that if both threads initially are enabled then they will also be initially enabled after composition. The composition of two LTSs is formally defined as follows:

Definition 5 ($\mathcal{T}_1 \| \mathcal{T}_2$)**.** *Given LTSs* $\mathcal{T}_1 = (S_1, A_1, \rightarrow_1, I_1)$ *and* $\mathcal{T}_2 = (S_2, A_2, \rightarrow_2, I_2)$ *such that* $S_1 = S_2$*, we define their* composition $\mathcal{T}_1 \| \mathcal{T}_2 = (S, A, \rightarrow, I)$ *where*

- $S = S_1 = S_2$
- $A = A_1 \cup A_2$
- $\rightarrow = \rightarrow_1 \cup \rightarrow_2$
- $I = I_1 \cap I_2$

Notice that composition is commutative and associative.

Execution traces of the LTSs are infinite sequences of states. Each state in the trace can be reached by a transition from the previous state, or, if there are no actions enabled, it is the same as the previous state.

Definition 6 (Trace). *Given an LTS* $\mathcal{T} = (S, A, \rightarrow, I)$*, we say that the infinite sequence* $x = x_0\, x_1\, x_2 \ldots$ *of states is a* trace *of* $\mathcal{T}$*, written* $\mathsf{trace}\,\mathcal{T}\,x$*, if*

- $x_0 \in I$ *and*
- *for all* i*, if there exists an* $a \in A$ *such that* $\mathsf{enabled}\,\mathcal{T}\,x_i\,a$*, then there exists an* $a' \in A$ *such that* $x_i \xrightarrow{a'} x_{i+1}$*, otherwise* $x_i = x_{i+1}$

We say a finite sequence $x_0, \ldots, x_k$ *is an* initial trace segment up to k*, denoted* $\mathsf{trace\_upto}\,\mathcal{T}\,x\,k$*, if for all* i *such that* $0 \leq i < k$ *the sequence* $x_0, \ldots, x_i$ *satisfies the conditions above.*

If there are no more transitions enabled we say a trace is *stuttering*. In particular we use $\mathsf{stutters}\,x_i$ to denote $\forall j.\ i \leq j \Rightarrow x_j = x_i$ and $\mathsf{stutters\_upto}\,x_i\,k$ to denote $\forall j.\ i \leq j \wedge j \leq k \Rightarrow x_j = x_k$. We also adopt the notation of Emerson [48] such that x^j denotes the suffix trace $x_j, x_{j+1}, x_{j+2}, \ldots$ and x_i^j denotes the segment trace $x_i, x_{i+1}, \ldots, x_{j-1}, x_j$.

We use $\mathsf{trace\_pred}\ \mathcal{T}\ x$ to denote that the infinite sequence satisfies the second condition of the definition above. Notice that we have

$$\mathsf{trace}\ \mathcal{T}\ x \Leftrightarrow x_0 \in I \wedge \mathsf{trace\_pred}\ \mathcal{T}\ x$$

A generalisation of traces is sometimes used, denoted $\mathsf{trace\_q}\ q\ \mathcal{T}\ x$, where the first state is required to satisfy predicate q. Hence we write

$$\mathsf{trace\_q}\ q\ \mathcal{T}\ x \Leftrightarrow q(x_0) \wedge \mathsf{trace\_pred}\ \mathcal{T}\ x$$

Notice that this immediately gives us

$$\mathsf{trace}\ \mathcal{T}\ x \Leftrightarrow \mathsf{trace\_q}\ (\in I)\ \mathcal{T}\ x$$

Finally we assume that all executions are fair, *i.e.* if a transition is enabled then it will eventually happen, otherwise it will become disabled. Note that in the literature this is usually referred to as "weak" fairness.

Definition 7 (Fairness). *Given an LTS $\mathcal{T}$ and an infinite sequence x such that* $\mathsf{trace}\ \mathcal{T}\ x$*, we say x is* fair *if for all i and a such that* $\mathsf{enabled}\ \mathcal{T}\ x_i\ a$*, there exists a $j \geq i$ such that either* $\neg\mathsf{enabled}\ \mathcal{T}\ x_j\ a$ *or* $x_j \xrightarrow{a} x_{j+1}$.

All definitions have been formalised in Isabelle/HOL. Figure 3.1 shows how we formalise $\mathsf{trace}\ \mathcal{T}\ x$ in Isabelle/HOL. States `s` are of type `'s`, whereas labels `a` are of type `'a`. An LTS is represented by a record, *i.e.* a collection of fields. A record is declared as follows:

$$\texttt{record}\ (\alpha_1, \ldots, \alpha_l)\ \mathtt{u} = (\tau_1, \ldots, \tau_m)\ \mathtt{v} + \mathtt{c}_1 :: \sigma_1 \ldots \mathtt{c}_n :: \sigma_n$$

Here, each α_i is a distinct type variable and each σ_j is a type containing at most the variables from $\alpha_1, \ldots, \alpha_l$. The type constructor `u` is new, whereas `v` specifies an existing record type. The n distinct field names are given by $\mathtt{c}_1, \ldots, \mathtt{c}_n$. The above definition introduces a new record type $(\alpha_1, \ldots, \alpha_l)\ \mathtt{u}$ by extending the (optional) existing record $(\tau_1, \ldots, \tau_m)\ \mathtt{v}$ with fields $\mathtt{c}_1 :: \sigma_1, \ldots, \mathtt{c}_n :: \sigma_n$. In our formalism the record `LTS` is not an extension of an existing record, but simply a collection of fields. The field `trans` of the record `LTS` represents the transition relation. For each tuple `(s,a,t)` where `s` and `t` are states, and `a` is a label, `trans` says whether there is a transition. The field `init` is a predicate on states and distinguishes the initial states. A trace `x` is an infinite list of states: we model this list as a function from the natural numbers to states. For example `x 0` denotes the initial state of trace `x`. The definitions for `enabled`, `stutters`, `trace_pred` and `trace` follow from those discussed previously.

Several useful results have been proved using our formalism. The most interesting of these is that any initial trace fragment can be extended to a full trace fragment by arbitrarily picking enabled transitions until no transitions are enabled anymore. This is exhibited in the following lemma which states that for every x that is an initial trace fragment up to j we can find a trace x' that coincides with x on the first j elements.

Lemma 1. $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j \Rightarrow \exists x'.\ \mathsf{trace\_pred}\ \mathcal{T}\ x' \wedge (\forall i.\ i < j \Rightarrow x_i = x'_i)$

```
record ('s, 'a) LTS =
    trans :: "'s * 'a * 's => bool"
     init :: "'s => bool"

types
   's trace = "nat => 's"

constdefs
   enabled :: "('s, 'a) LTS => 's => 'a => bool"
   enabled lts s a == (EX t. trans lts (s,a,t))"

   stutters :: "'s trace => nat => bool"
   stutters x i == (ALL j. i <= j --> x j = x i)"

   trace_pred :: "('s, 'a) LTS => 's trace => bool"
   trace_pred lts x ==
       (ALL i. (if (EX a. enabled lts (x i) a)
                then (EX a. trans lts (x i, a, x (Suc i)))
                else stutters x i))"

   trace :: "('s, 'a) LTS => 's trace => bool"
   trace lts x == init lts (x 0) & trace_pred lts x"
```

Figure 3.1: Isabelle/HOL formalism of trace $\mathcal{T}\,x$.

Note that the proofs of all lemmata in this chapter are given in Section 3.7. Some of these can be quite long-winded, which is why we prefer to put them in a separate section.

3.2.2 Modelling Java

As mentioned above our program model is inspired by the programming language Java. At any given time Java's virtual machine can support multiple threads of execution. These threads independently execute code that operates on values and objects residing in a shared main memory. Each thread has a working memory in which it keeps working copies of the values of variables from the main memory. Following the Java language specification [61], threads execute code by performing a sequence of atomic memory actions. These memory actions can be lock, unlock, write, read, store, load, assign and use; all with appropriate parameters. These actions cannot occur in an arbitrary order: a set of rules restrict the possible interactions; see [61; 29] for details. Typically we would assume Java's memory actions are the labels of our transitions – the state space is modelled as a mapping from variables to values – however we never make this explicit. Moreover, since each action of the Java memory model is atomic we assume that identically labelled transitions have identical effects on the state space.

$$s \xrightarrow{a} t \wedge s' \xrightarrow{a} t' \Rightarrow \forall x.\, s(x) \neq t(x) \wedge s(x) = s'(x) \Rightarrow s'(x) \neq t'(x) \qquad \text{(Ass 1)}$$

⟨Pattern⟩ = Universal ⟨Prop⟩ ⟨Scope⟩
| Absent ⟨Prop⟩ ⟨Scope⟩
| Exists ⟨Prop⟩ ⟨Scope⟩
| ⟨Prop⟩ RespondsTo ⟨Prop⟩ ⟨Scope⟩
| ⟨Prop⟩ Precedes ⟨Prop⟩ ⟨Scope⟩

⟨Scope⟩ = Globally
| After ⟨Prop⟩
| Before ⟨Prop⟩
| Between ⟨Prop⟩ ⟨Prop⟩
| AfterUntil ⟨Prop⟩ ⟨Prop⟩

Figure 3.2: Syntax of specification patterns

This assumption states that if we have two identically labelled transitions $s \xrightarrow{a} t$ and $s' \xrightarrow{a} t'$, then if s and s' coincide on x, then if the first transition will change the value of x (*i.e.* $s(x) \neq t(x)$), then the second transition will also change the value of x, thus $s'(x) \neq t'(x)$.

In fact, in Java, each thread also has a private memory. We could explicitly incorporate this, but this is not strictly necessary: it is sufficient to assume that certain parts of the global memory will only be changed by a single thread. We assume that all threads are already created and can be represented by a single LTS. Our model would allow dynamic thread creation, however this would make the separation into different threads more involved.

3.3 Temporal formulae

As a property specification language we use the specification patterns originally proposed by the Bandera team at Kansas State University [46; 133]. We have previously discussed these patterns in Section 2.3 of Chapter 2. Specification patterns describe the most common constructs found in temporal logic specifications. For each pattern a mapping into different logics (such as LTL [48], CTL [48] and regular alternation-free μ-Calculus [103]) is defined. Each pattern describes a property that has to hold in a certain region of the system execution. This region is called the scope of the pattern. Two kinds of patterns are distinguished: occurrence patterns (absence, universality and existence) and order patterns (response and precedence). Figure 3.2 shows the syntax of the specification patterns used in our work.

To give a semantics to these specification patterns, we use the mapping of the patterns into LTL as defined on the specification patterns website [133]. We call this mapping pat2ltl and refer to it as the "pattern mapping of ϕ". Figure 3.3 presents the specification pattern mappings into LTL. We use a direct translation into LTL following [48] so that we may formalise the semantics of the specification patterns into Isabelle/HOL. (Note that the semantics of the temporal specification language described in Chapter 2, although based on LTL, did not require

Universal p Globally	$\Box p$
Universal p After q	$\Box(q \rightarrow \Box p)$
Universal p Before r	$\Diamond r \rightarrow (p \,\mathsf{U}\, r)$
Universal p Between $q\,r$	$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (p \,\mathsf{U}\, r))$
Universal p AfterUntil $q\,r$	$\Box((q \wedge \neg r) \rightarrow (p \,\mathsf{W}\, r))$
Absent p Globally	$\Box \neg p$
Absent p After q	$\Box(q \rightarrow \Box \neg p)$
Absent p Before r	$\Diamond r \rightarrow (\neg p \,\mathsf{U}\, r)$
Absent p Between $q\,r$	$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p \,\mathsf{U}\, r))$
Absent p AfterUntil $q\,r$	$\Box((q \wedge \neg r) \rightarrow ((\neg p) \,\mathsf{W}\, r))$
Exists p Globally	$\Diamond p$
Exists p After q	$\Box \neg q \vee \Diamond(q \wedge \Diamond p)$
Exists p Before r	$\neg r \,\mathsf{W}\, (p \wedge \neg r)$
Exists p Between $q\,r$	$\Box((q \wedge \neg r) \rightarrow (\neg r \,\mathsf{W}\, (p \wedge \neg r)))$
Exists p AfterUntil $q\,r$	$\Box((q \wedge \neg r) \rightarrow (\neg r \,\mathsf{U}\, (p \wedge \neg r)))$
p Precedes q Globally	$\neg q \,\mathsf{W}\, p$
p Precedes q After r	$\Box \neg r \vee \neg r \,\mathsf{W}\, (r \wedge (\neg q \,\mathsf{W}\, p))$
p Precedes q Before r	$\Diamond r \rightarrow (\neg q \,\mathsf{U}\, (p \vee r))$
p Precedes q Between $r\,s$	$\Box((r \wedge \neg s \wedge \Diamond s) \rightarrow (\neg q \,\mathsf{U}(p \vee s)))$
p Precedes q AfterUntil $r\,s$	$\Box((r \wedge \neg s) \rightarrow (\neg q \,\mathsf{W}\, (p \vee s)))$
p RespondsTo q Globally	$\Box(q \rightarrow \Diamond p)$
p RespondsTo q After r	$\Box(r \rightarrow \Box(q \rightarrow \Diamond p))$
p RespondsTo q Before r	$\Diamond r \rightarrow (q \rightarrow (\neg r \,\mathsf{U}\, (p \wedge \neg r))) \,\mathsf{U}\, r$
p RespondsTo q Between $r\,s$	$\Box((r \wedge \neg s \wedge \Diamond s) \rightarrow (q \rightarrow (\neg s \,\mathsf{U}\, (p \wedge \neg s))) \,\mathsf{U}\, s)$
p RespondsTo q AfterUntil $r\,s$	$\Box((r \wedge \neg s) \rightarrow ((q \rightarrow (\neg s \,\mathsf{U}\, (p \wedge \neg s))) \,\mathsf{W}\, s))$

Figure 3.3: Specification pattern mappings into LTL

such a direct translation.) We say $\mathcal{T}$ satisfies formula ϕ, denoted $\mathcal{T} \models \phi$, if for all infinite sequences x such that $\mathsf{trace}\,\mathcal{T}\,x$ we have $x \models_{\mathsf{LTL}} \mathsf{pat2ltl}(\phi)$, where $\models_{\mathsf{LTL}}$ corresponds to the usual satisfaction relation on LTL formulae. Alternatively, we sometimes write $\mathcal{T}, q \models \phi$ if all traces starting with property q satisfy the formula ϕ, *i.e.* for all infinite sequences x such that $\mathsf{trace\_q}\,q\,\mathcal{T}\,x$ we have $x \models_{\mathsf{LTL}} \mathsf{pat2ltl}(\phi)$. We call this *q-satisfaction.* Notice that to show that a pattern ψ with scope $\mathsf{Between}\,q\,r$ holds on $\mathcal{T}$ it is sufficient to show that $\mathcal{T}$ q-satisfies $\psi\,\mathsf{Between}\,q\,r$.

$$\mathcal{T}, q \models \psi\,\mathsf{Between}\,q\,r \Rightarrow \mathcal{T} \models \psi\,\mathsf{Between}\,q\,r \tag{3.1}$$

Similar results can be proven for the scopes $\mathsf{After}\,q$ and $\mathsf{AfterUntil}\,q\,r$.

While formalising the semantics we found some small ambiguities in the mappings (missing brackets, *etc.*) and we encountered one significant problem. Following the website the pattern $p\,\mathsf{Precedes}\,q\,\mathsf{After}\,r$ is mapped into the LTL formula $\Box\neg r \vee \Diamond(r \wedge (\neg q\,\mathsf{W}\,p))$ where W is the weak until operator, *i.e.* $p\,\mathsf{W}\,q$ means p holds until q, or p holds forever. This formula says that either r never holds, or there is a place where r holds, and from that place onwards q will not hold unless p has held before. However notice that this mapping does not require that the property $\neg q\,\mathsf{W}\,p$ holds the first time r is true, it only requires it to hold sometime that r is true. Thus it would accept the trace below, the reason being that because the second time r is true, the property $\neg q\,\mathsf{W}\,p$ holds (since in the next state p holds and only one state later q becomes true). We assume only the properties mentioned in the states hold, all other properties are false.

In our opinion this trace should be considered incorrect because after the first occurrence of r, q occurs without a preceding occurrence of p. Therefore we changed the mapping of $p\,\mathsf{Precedes}\,q\,\mathsf{After}\,r$ into $\Box\neg r \vee \neg r\,\mathsf{W}\,(r \wedge (\neg q\,\mathsf{W}\,p))$ which rejects the trace above and corresponds better to our intuition of the meaning of this pattern. Moreover this closely resembles the mapping of this pattern into a CTL formula.

3.4 The factorisation rules

As explained above our aim is to factorise the verification of temporal properties over the different threads of a program. Given a program and a temporal property, we divide the different threads in the program into two groups: for one group we show that they establish the property, for the other group we show that they do not affect the property. We assume that each thread is modelled by a labelled transition system and that a program consists of a collection of threads. Since our LTS composition operator is associative and commutative, it is sufficient to provide factorisation rules for the composition of two LTSs $\mathcal{T}_1 \| \mathcal{T}_2$.

Preservation To show that an LTS does not affect a temporal property we require that it does not change (*i.e.* it preserves) any variable that is related to the property. Typically these related variables will be all the variables that are mentioned in the property and any variable on which these variables (directly or indirectly) depend. However, in general, it is not necessary that the

second LTS always preserves the set of variables. For each temporal property we can state the precise conditions under which this set of variables has to be preserved. Moreover, in some cases the second thread might also make a step which does not preserve the set of variables but actually makes the temporal property hold for the composed system. For example, to show that a temporal property Exists p Globally holds on a composed system, *i.e.* on every path there always exists a p, it is sufficient to show that the second (group of) thread(s) preserves the set of variables on which p depends, unless it makes p true. To be as general as possible, our factorisation rules allow the second thread, wherever possible, to make the property hold for the composed system.

Before formally defining preservation we first define equality of states with respect to a set of variables V. As explained in Section 3.2 we consider states as mappings from variables to values. We say two states are *V-equal* if they coincide on the values of all variables in V.

Definition 8 (V-equality). *Given states s and t, and a set of variables V we define* V-equality *between s and t, denoted $s =_V t$, as follows:*

$$s =_V t \Leftrightarrow \forall v.\ v \in V \Rightarrow s(v) = t(v)$$

Below is listed a number of trivially proven properties attributable to V-equality. They include reflexivity, symmetry and transitivity. Note $\mathcal{V}$ denotes the set of *all* variables.

$$\begin{array}{rcl} & & s =_V s \\ s =_V t & \Leftrightarrow & t =_V s \\ s =_V t \Rightarrow t =_V u & \Rightarrow & s =_V u \\ s =_V t \wedge W \subseteq V & \Rightarrow & s =_W t \\ s =_\emptyset t & \Leftrightarrow & \mathsf{true} \\ s =_\mathcal{V} t & \Leftrightarrow & s = t \end{array}$$

Now we are ready to define preservation.

Definition 9 (Preserves). *Given an LTS $\mathcal{T}$, a set of variables V, and state predicates p and q, we define* preservation *as follows:*

$$p \models \mathcal{T} \text{ preserves } V \mid q \quad \Leftrightarrow \quad \forall s\, t\, a.\ \ p(s) \wedge s \xrightarrow{a} t \ \Rightarrow\ (s =_V t \wedge p(t)) \vee q(t)$$

Thus when p holds in a state s all states that are directly reachable from this state should either preserve V, or make q hold. We say (informally) that "$\mathcal{T}$ preserves V until q holds". Notice that it is easy to prove that preservation is preserved by the subset relation.

$$U \subseteq V \Rightarrow p \models \mathcal{T} \text{ preserves } V \mid q \Rightarrow p \models \mathcal{T} \text{ preserves } U \mid q$$

Here we do not go further into how the preservation property can be checked, but we believe that this can be done relatively easily using similar standard techniques for program verification; for example those used to check assignable clauses of JML specifications as implemented in the Chase tool discussed in Section 2.2.1.

Dependency sets As explained above, for each temporal property we define the set of variables that have to be preserved by the various threads of a program. This set of variables depends on the variables used in the different state properties and on the program, or more accurately, on the program's dependence graph.

Recall that a standard Program Dependence Graph, or PDG, is a directed graph with nodes corresponding to statements and control predicates, and edges corresponding to data and control dependencies. Such dependencies are found by constructing a Control Flow Graph, or CFG. A CFG is comprised of nodes which represent each statement and control predicate in a program; they also include nodes *start* and *stop* corresponding to the beginning and end of a program, respectively. An edge from node n to m indicates the possible flow of control from n to m. A node n is said to dominate a node m in a CFG if every path from *start* to m passes through n. A node n is said to be post-dominated by m in a CFG if all paths from n to *stop* pass through m. A node n is data dependent on node m if m assigns a value to a variable v that is referenced at n, and there is a path between both nodes that does not contain any other assignments to v. A node n is control dependent on m if m branches and there is one path that passes through n, and one path that does not.

In [64; 47] the different dependencies that can occur in a multi-threaded Java program are identified. Besides the traditional data and control dependencies, four additional dependencies are used to define the edges of a multi-threaded Java program's PDG: interference, divergence, synchronization and ready dependencies. As summarised in [85] a node n is interference dependent on a node m of another thread if m assigns a value to a variable v that is referenced at n. A node n is divergence dependent on m if m is a pre-divergence point and m dominates n. (A pre-divergence point is the point of a `for` or `while` loop where the condition is checked to either leave or stay in the loop.) A node n is synchronization dependent on m if m is one of the statements that define the synchronization block that contains n. (A synchronization block describes any code within a `synchronized` method; this is the method used to acquire and release a lock.) A node n is ready dependent on m if m is one of the statements that can release a lock that is needed to reach n.

The notion of a PDG that we use as a basis for our work here is a standard PDG (as described in [114]) which incorporates the traditional data and control dependencies along with the four additional dependencies defined in [64; 47]. We use $\mathsf{dep}_p\,\mathcal{T}$ to denote the set of variables on which the variables in p depend with respect to the program represented by $\mathcal{T}$. To be more precise: suppose the variables in p are declared at nodes $\{n_i, n_j, n_k\}$ of the program's PDG. Call this set N. Then the transitive closure N' of the dependencies in the PDG can be computed with respect to each node in N. The variables declared or referenced at the nodes of N' hence form the dependency set $\mathsf{dep}_p\,\mathcal{T}$. Echoing [120], we assume that our dependency analysis distributes over conjunction and negation. This is exhibited in the following assumptions.

$$\mathsf{dep}_{p \wedge q}\,\mathcal{T} = \mathsf{dep}_p\,\mathcal{T} \cup \mathsf{dep}_q\,\mathcal{T} \qquad \text{(Ass 2)}$$

$$\mathsf{dep}_{\neg p}\,\mathcal{T} = \mathsf{dep}_p\,\mathcal{T} \qquad \text{(Ass 3)}$$

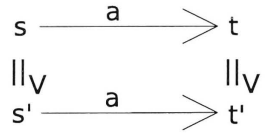

Figure 3.4: Illustration of assumption 4

The intuition behind the latter assumption is that a predicate p cannot be falsified by a program only updating variables outside of p's dependency set. For convenience we also write $\mathsf{dep}_{p,q}\,\mathcal{T}$ to denote $\mathsf{dep}_p\,\mathcal{T} \cup \mathsf{dep}_q\,\mathcal{T}$.

Furthermore we assume that the dependency relation properly coincides with the transition relation: by this we mean that for any transition $s \xrightarrow{a} t$ that does not preserve the variables in the dependency set, and for any state s' that is equivalent to s with respect to the dependency set, we can find a state t' that can be reached with a similarly labelled transition and which is also equivalent to t with respect to the dependency set. (See Figure 3.4 for an illustration.) This ensures that the dependency set is closed and that there are no "leaking" dependencies. In other words, if we were to reduce the state space to the set V then the transitions would be indistinguishable. We specify this assumption formally as follows:

$$s \xrightarrow{a} t \Rightarrow s \neq_{\mathsf{dep}_p\,\mathcal{T}} t \Rightarrow \forall s'.s =_{\mathsf{dep}_p\,\mathcal{T}} s' \Rightarrow \exists t'.s' \xrightarrow{a} t' \wedge t =_{\mathsf{dep}_p\,\mathcal{T}} t' \qquad \text{(Ass 4)}$$

A last important assumption concerning dependency sets is given below. It tells us that if two states s and t are equivalent with respect to the set $\mathsf{dep}_p\,\mathcal{T}$ then the satisfiability of p in s implies the satisfiability of p in t and *vice versa* (which follows directly from the symmetry of the V-equality relation).

$$p(s) \wedge s =_{\mathsf{dep}_p\,\mathcal{T}} t \Rightarrow p(t) \qquad \text{(Ass 5)}$$

This is an entirely reasonable assumption to make, because if s and t are $(\mathsf{dep}_p\,\mathcal{T})$-equivalent, then for all $x \in \mathsf{dep}_p\,\mathcal{T}$ we have $s(x) = t(x)$. Since $\mathsf{dep}_p\,\mathcal{T}$ denotes the set of all variables on which the variables in p depend, we expect that if p holds on s then p will also hold on t.

The rules Finally we are ready to present the different rules. In the remainder of this section we present the intuition for two rules, Exists p Between $q\,r$ and p RespondsTo q After r. We then discuss a compromise we were forced to make for two of our rules, Universal p Before r and Universal p Between $q\,r$. The next section discusses our verification strategy in depth and presents the proof of the Exists p Between $q\,r$ rule in more detail. We ask that the reader refer to Appendix A for an overview of the rules for universality, absence, existence, precedence and response. Note that the rules for absence can be directly derived from the rules for universality by using the following equivalence (where s denotes an arbitrary scope).

$$\mathcal{T} \models \mathsf{Absent}\, p\, s \Leftrightarrow \mathcal{T} \models \mathsf{Universal}\, \neg p\, s$$

Exists p Between $q\, r$ This rule states that in order to show a composed $\mathcal{T}_1 \| \mathcal{T}_2$ system satisfies Exists p Between $q\, r$ – meaning that on all traces of $\mathcal{T}_1 \| \mathcal{T}_2$ after a state where q is true , if r eventually holds, then p holds earlier – then the following premises are sufficient: (1) every trace in $\mathcal{T}_1$ that starts with a state satisfying q satisfies the pattern mapping of the property, *i.e.* $\mathcal{T}_1$ q-satisfies Exists p Between $q\, r$, and (2) $\mathcal{T}_2$ preserves the properties p and r until p is true.

$$\frac{\mathcal{T}_1, q \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,r}\, \mathcal{T}_1) \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r}$$

The intuition behind this is as follows. If we want to show that a composed $\mathcal{T}_1 \| \mathcal{T}_2$ system satisfies a property Exists p Between $q\, r$, we need to show that the pattern mapping of Exists p Between $q\, r$ holds on all traces of $\mathcal{T}_1 \| \mathcal{T}_2$. So suppose we have a trace of $\mathcal{T}_1 \| \mathcal{T}_2$ in which there is a state where q is true. If r eventually holds on this trace, then we need to show that p holds earlier. If r does not hold eventually, then the property trivially holds. The second premise of the proof rule tells us that any trace of $\mathcal{T}_1 \| \mathcal{T}_2$ between states satisfying q and then r can be considered equivalent to a trace of $\mathcal{T}_1$ because $\mathcal{T}_2$ is not allowed to interfere throughout this segment, *i.e.* as long as p and r are not true, $\mathcal{T}_2$ will not change their validity. (There is one exception to this which we will address shortly.) Since the first premise states that the pattern mapping of Exists p Between $q\, r$ holds on any trace of $\mathcal{T}_1$ starting with q, we conclude that the pattern mapping of Exists p Between $q\, r$ holds on the trace of the composed $\mathcal{T}_1 \| \mathcal{T}_2$ system. The exception that we remarked upon earlier occurs when $\mathcal{T}_2$ itself makes p true. In this case the dependency sets of p and r no longer have to be preserved because it is sufficient to have a single existence of p, thus the formula on the composed system trivially holds.

p RespondsTo q After r Similarly, if we wish to show that a system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies a property p RespondsTo q After r – meaning that on all traces of $\mathcal{T}_1 \| \mathcal{T}_2$ after a state where r is true, if q holds somewhere after r, then p eventually holds – then the following premises are sufficient: (1) any $\mathcal{T}_1$ trace starting with a state satisfying r satisfies the pattern mapping of p RespondsTo q After r, and (2) as long as $\neg p$ holds, $\mathcal{T}_2$ preserves p and q unless it makes p true.

$$\frac{\mathcal{T}_1, r \models p\ \mathsf{RespondsTo}\, q\, \mathsf{After}\, r \quad \neg p \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,q}\, \mathcal{T}_1) \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p\ \mathsf{RespondsTo}\, q\, \mathsf{After}\, r}$$

The validity of this rule can be intuitively understood as follows. Suppose we have a trace of $\mathcal{T}_1 \| \mathcal{T}_2$ in which there is a state where r is true. We need to show that the pattern mapping of p RespondsTo q After r holds on this trace, *i.e.* if q holds somewhere after r, then we need to show that eventually p will hold. The second premise tells us that any trace of $\mathcal{T}_1 \| \mathcal{T}_2$ after a state satisfying r can be considered equivalent to a $\mathcal{T}_1$ trace because $\mathcal{T}_2$ does not interfere by changing any of the relevant variables. The first premise tells us

that any $\mathcal{T}_1$ trace starting with r satisfies the pattern mapping of p RespondsTo q After r. Hence we can conclude that the pattern mapping of p RespondsTo q After r holds on the trace of the composed system. As in the example above, an exception to $\mathcal{T}_2$'s non-interference is that it can also make p hold, in which case the formula on the composed system trivially holds.

Problems We would like to be able to prove the validity of the following rule since it corresponds favourably with our intuitive understanding of the approach. Namely, to show that a composed $\mathcal{T}_1 \| \mathcal{T}_2$ system satisfies a property Universal p Before r the following premises are sufficient: (1) every trace in $\mathcal{T}_1$ satisfies the pattern mapping of the property, and (2) $\mathcal{T}_2$ preserves the properties p and r until r is true.

$$\frac{\mathcal{T}_1 \models \text{Universal } p \text{ Before } r \quad p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } (\text{dep}_{p,r}\, \mathcal{T}_1) \mid r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \text{Universal } p \text{ Before } r}$$

However we cannot prove the validity of this rule; the reasoning behind it is thus. Suppose we have a trace of the composed $\mathcal{T}_1 \| \mathcal{T}_2$ system. To prove the rule's validity, we need to show (using our assumptions) that the pattern mapping of Universal p Before r holds on this trace. The pattern mapping of Universal p Before r is $\Diamond r \rightarrow (p \,\mathsf{U}\, r)$. Here U is a strong until: r must be true at some stage in order for p to hold previously. Now our second premise tells us: (1) that no $\mathcal{T}_2$ transition can ever make r true or p false because $\mathcal{T}_2$ always preserves r and p's dependent set; and (2) until r holds, any trace of $\mathcal{T}_1 \| \mathcal{T}_2$ can be considered equivalent to a $\mathcal{T}_1$ trace because $\mathcal{T}_2$ does not change any of the relevant variables. The first premise tells us that p is universally true before r if r is made true by a $\mathcal{T}_1$ transition. Hence we can conclude that p is universally true before r on the composed trace if r is made true by a $\mathcal{T}_1$ transition. However there is no guarantee that $\mathcal{T}_1$ will make r true at all. If r is never made true by a $\mathcal{T}_1$ transition, the only way we can prove this rule is if we force $\mathcal{T}_2$ to forever preserve the dependent sets of p and r. After changing the preservation statement to reflect this, we therefore have the following (valid) rule:

$$\frac{\mathcal{T}_1 \models \text{Universal } p \text{ Before } r \quad \neg r \models \mathcal{T}_2 \text{ preserves } (\text{dep}_{p,r}\, \mathcal{T}_1) \mid \text{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \text{Universal } p \text{ Before } r}$$

The premise of this rule tells us that while r is not true, any $\mathcal{T}_1 \| \mathcal{T}_2$ trace can be considered equivalent to a $\mathcal{T}_1$ trace because $\mathcal{T}_2$ never changes any of the relevant variables. Since we can show that the pattern mapping holds on a $\mathcal{T}_1$ trace using our first premise, we can show that the pattern mapping holds on the composed trace.

The same argument works for the Universal p Between $q\, r$ rule with the (more intuitive) preservation statement $p \wedge \neg r \models \mathcal{T}_2$ preserves $(\text{dep}_{p,r}\, \mathcal{T}_1) \mid r$. Because the pattern mapping of Universal p Between $q\, r$ features a strong until, the preservation statement needs to be changed to $\neg r \models \mathcal{T}_2$ preserves $(\text{dep}_{p,r}\, \mathcal{T}_1) \mid \text{false}$. Now contrast this with the rule Universal p AfterUntil $q\, r$ given below.

$$\frac{\mathcal{T}_1, q \models \text{Universal } p \text{ AfterUntil } q\, r \quad p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } (\text{dep}_{p,r}\, \mathcal{T}_1) \mid r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \text{Universal } p \text{ AfterUntil } q\, r}$$

Again, a $\mathcal{T}_2$ transition never makes r true. However if $\mathcal{T}_1$ also does not make r true p will still hold because the pattern mapping of AfterUntil features a weak until, *viz.* $\Box((q \wedge !r) \rightarrow (p \mathsf{W} r))$. If $\mathcal{T}_1$ does make r true then $\mathcal{T}_2$ just preserves the dependent set until this $\mathcal{T}_1$ transition. Hence in this particular case, no adjustments need to be made to the preservation statement.

3.5 Formalisation and correctness

As mentioned previously, all twenty-five rules have been formalised and proven correct using Isabelle/HOL [109]. We have relied on a semantics/interpretation of Bandera's specification patterns [133] in LTL. Furthermore we have assumed:

- that we can represent a Java program as a Labelled Transition System (LTS)
- the existence of a standard Program Dependence Graph (PDG) that provides a notion of dependency sets
- that the PDG satisfies assumptions 2-5
- that the LTS satisfies assumptions 1 and 6

This section examines the correctness proof of one of our rules rule – Exists Between – in more detail. Similar proofs have been constructed for all other rules. However, before we begin, we will first sketch the general approach we used for verification of the rules.

Suppose C_1, C_2 and C_3 are arbitrary boolean formulae containing the atomic propositions in ϕ. A generalised rule is given below.

$$\frac{\mathcal{T}_1 \models \phi \qquad C_1 \models \mathcal{T}_2 \text{ preserves } (\mathsf{dep}_{C_2}\ \mathcal{T}_1)\,|\,C_3}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

In order to show that a rule of this form is correct, we have to show that for an arbitrary trace of $\mathcal{T}_1 \| \mathcal{T}_2$ the property $\mathsf{pat2ltl}(\phi)$ holds. Intuitively, if we have a trace x of a composed LTS we should be able to find an equivalent trace of the isolated $\mathcal{T}_1$ system because $\mathcal{T}_2$ is guaranteed not to affect the variables relevant to the property. (In many cases, such as the rule Exists Between, it suffices to show that there is an equivalent initial segment.) However x can contain arbitrary transitions that are irrelevant with respect to the property ϕ. These transitions can be made both by $\mathcal{T}_1$ and $\mathcal{T}_2$. Therefore we use slicing to construct (an initial fragment of) a trace y from x, and we show that y is (an initial fragment of) a trace of $\mathcal{T}_1$. Since we know that $\mathcal{T}_1$ satisfies ϕ and $\mathcal{T}_2$ preserves the appropriate set of variables, we can conclude that $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies ϕ.

The main challenge in the verification of the different rules is the proof that one can construct the (initial fragment of a) trace of $\mathcal{T}_1$. Essentially we need to show that if a state s is enabled in the sliced $\mathcal{T}_1$ trace, then a transition is made otherwise the trace stutters. We relate s to a state s_o in the original trace and consider the different possibilities for s_o in the original trace: s_o is stuttering; s_o makes a transition to a state that is equivalent with respect to the slicing relation, which is thus not visible in the sliced trace; or s_o makes a transition to a state that is different with respect to the slicing relation, which is thus visible in the sliced trace. In the latter case, we have to distinguish whether the transition is made by $\mathcal{T}_1$ or $\mathcal{T}_2$ (a $\mathcal{T}_2$ transition

usually leads to a contradiction). These proofs are long and involved and we will not go into further details here.

We first introduce appropriate definitions and assumptions that we use for slicing. We then discuss the proof of the rule Exists Between in more detail.

Slicing As explained above, we use the notion of slicing to remove irrelevant transitions from our LTSs. Additionally, we define a function that takes a normal trace as input and outputs a trace of the sliced LTS. As mentioned previously in Section 3.1, slicing techniques are usually applied to code, removing those parts of a program that are irrelevant to the property being verified. The sliced program is then modelled by a smaller, more manageable transition system; see [64] for an automated approach to slicing Java programs. In our model however, rather than slicing the program, we slice the transition systems themselves.

In our verifications we assume that if we want to show that an LTS $\mathcal{T}$ satisfies a property p, then slicing that system with respect to $\mathsf{dep}_p\,\mathcal{T}$ will not change the system's ability to satisfy p. We do not formally prove this, but we refer to [64] for a proof that properties (expressed in LTL) are preserved by slicing. Below we will show the formalisation of this assumption.

The following definitions show how we slice states, LTSs and traces. Note that for all slicing operations we fix a set of variables V on which the property ϕ depends. Also recall that states are defined as mappings from variables to values; a sliced state is a restriction of this mapping to the variables in V. Some further asides: since all functions in Isabelle/HOL have to be total, we map all other variables to some unknown constant arbitrary; moreover the lambda notation is used in Isabelle to describe functions. This notation provides a means of referring to functions without actually having to name them. Suppose $f : S \to S'$ is a function which for any element $x \in S$ returns a value $f(x)$ which can be exactly described by the expression e (usually involving x), then we write $\lambda x.e$ for the function f.

Definition 10 (Sliced state). *Given a state s and a set of variables V, we define the* sliced state $s_{|V}$ *as follows:*

$$s_{|V} = \lambda v.\ \mathsf{if}\ v \in V\ \mathsf{then}\ s(v)\ \mathsf{else}\ \mathsf{arbitrary}$$

Notice that we can immediately prove the following results for sliced states.

$$\begin{array}{rcl} s =_V t & \Leftrightarrow & s_{|V} = t_{|V} \\ (s_{|V})_{|V'} & = & s_{|V \cap V'} \\ V \subseteq V' & \Rightarrow & (s_{|V'})_{|V} = s_{|V} \\ & & s_{|V} =_V s \end{array}$$

For convenience, given an arbitrary boolean state predicate C, we use $C_{|V}$ to denote the sliced state predicate $\lambda s.\ \exists s'.\ C(s') \wedge s'_{|V} = s$.

Next we define a sliced LTS. The sliced transition relation is defined as a restriction of the original transition relation, only keeping the transitions that affect variables in V. Note that initial states are preserved by slicing.

Definition 11 (Sliced LTS). *Given an LTS $\mathcal{T} = (S, A, \to, I)$ we define a* sliced LTS *with respect to the set V, as* $\mathsf{slice}\,\mathcal{T}\,V = (S', A', \to', I')$, *where*

- $S' = \bigcup_{s \in S} s_{|V}$
- $A' = A$
- $\rightarrow' = \{(s', a, t') \mid \exists s\, t.\ s \xrightarrow{a} t \wedge s_{|V} = s' \wedge t_{|V} = t' \wedge s' \neq t'\}$
- $I' = \bigcup_{i \in I} i_{|V}$

As mentioned above, an important assumption of our model is the slicing assumption which states that an LTS $\mathcal{T}$ q-satisfies a temporal property *iff* $\mathsf{slice}\,\mathcal{T}\,V$ q-satisfies the same property. We formalise this assumption as follows:

$$\mathcal{T}, q \models \phi \Leftrightarrow \mathsf{slice}\,\mathcal{T}\,V, q_{|V} \models \phi \qquad \text{(Ass 6)}$$

Notice that q can be instantiated with $\lambda s.\ s \in I$ when using this assumption for temporal properties with scope Before or Globally.

In order to define a sliced trace, we use an auxiliary function $\mathsf{nrss}\,V\,x\,k$ which counts the number of different states (with respect to V) in the first k states of a trace x. This function is recursively defined by the following two equations.

$$\begin{array}{rcl} \mathsf{nrss}\,V\,x\,0 & = & 0 \\ \mathsf{nrss}\,V\,x\,(\mathsf{Suc}\,k) & = & (\mathsf{if}\ x_k =_V x_{\mathsf{Suc}\,k}\ \mathsf{then}\ 0\ \mathsf{else}\ 1) + \mathsf{nrss}\,V\,x\,k \end{array}$$

As an example, suppose the original trace is $x_0 x_1 \ldots x_n \ldots$ and suppose $x_0 =_V x_1$ and $x_1 =_V x_2$ and all other states are not V-equivalent. Then $\mathsf{nrss}\,V\,x\,n = n - 2$. Note this function is also monotonous in its last argument.

Lemma 2. $i \leq j \Rightarrow \mathsf{nrss}\,V\,x\,i \leq \mathsf{nrss}\,V\,x\,j$

In addition, if the number of sliced states up to i is the same as the number of sliced states up to j, then this implies x_i is V-equivalent to x_j.

Lemma 3. $\mathsf{nrss}\,V\,x\,i = \mathsf{nrss}\,V\,x\,j \Rightarrow x_i =_V x_j$

Notice that the converse may fail since an LTS can reach a single state several times.

Finally we are ready to define a sliced trace.

Definition 12 (Sliced trace). *Given a trace x we define a* sliced trace *with respect to V as follows:*

$$\begin{array}{rcl} \mathsf{slice\_trace}\,V\,x = \lambda i. & \mathsf{if}\ (\exists j.\ i \leq \mathsf{nrss}\,V\,x\,j) \\ & \mathsf{then}\ (x_{\mathsf{least}\,j.\ \mathsf{nrss}\,V\,x\,j = i})_{|V} \\ & \mathsf{else}\ (x_{\mathsf{least}\,j.\ \forall k.\ j \leq k \Rightarrow \mathsf{nrss}\,V\,x\,j = \mathsf{nrss}\,V\,x\,k})_{|V} \end{array}$$

If we want to know what the i^{th} state is in $\mathsf{slice\_trace}\,V\,x$, we first check whether there exists at least i different states in x with respect to V. If this is the case, then if j is the smallest number for which $\mathsf{nrss}\,V\,x\,j = i$, then we return x_j restricted to V. Otherwise there are less than i different states, thus the sliced trace is stuttering. Suppose x_j is the first state where

the stuttering begins, *i.e.* afterwards the number of sliced states remains constant, namely $\forall k.\, j \leq k \Rightarrow \mathsf{nrss}\, V\, x\, j = \mathsf{nrss}\, V\, x\, k$. Then we return x_j restricted to V.

To conclude we present two lemmata concerning sliced traces. The first relates the sliced trace with the function nrss. It says that for any i, if j is the number of different sliced states up to i, then the j^{th} state in the sliced trace is equal to x_i restricted to V.

Lemma 4. $(\mathsf{slice\_trace}\, V\, x)_{\mathsf{nrss}\, V\, x\, i} = (x_i)_{|V}$

The second property tells us that every state in the sliced trace can be related to a state in the original trace, *i.e.* for the i^{th} state in the sliced trace there exists a j such that this state is equal to x_j restricted to V and either the number of different sliced states up to j is equal to i, or the number of different sliced states is strictly less than i for any k. In the latter case the sliced trace is stuttering from the i^{th} state onwards.

Lemma 5. $\exists j.\, (\mathsf{slice\_trace}\, V\, x)_i = (x_j)_{|V} \wedge \ (\mathsf{nrss}\, V\, x\, j = i \vee \forall k.\, \mathsf{nrss}\, V\, x\, k < i)$

Verification of the rule Exists Between Next, we look at the proof of the Exists Between rule in detail.

$$\frac{\mathcal{T}_1,\, q \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,r}\, \mathcal{T}_1) \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r}$$

First we observe that because of Equation (3.1) on page 48, it is sufficient to show that $\mathcal{T}_1 \| \mathcal{T}_2$ q-satisfies this property. Second, we observe that the pattern Exists p Between $q\, r$ maps into the LTL formula $\Box((q \wedge \neg r) \rightarrow (\neg r \mathsf{W} (p \wedge \neg r)))$. Using the semantics of LTL, this tells us that we have to show the following:

$$\begin{aligned}\forall x.\, \mathsf{trace\_q}\, q\, (\mathcal{T}_1 \| \mathcal{T}_2)\, x \Rightarrow \forall i.\, (q(x_i) \wedge \neg r(x_i) \Rightarrow \\ (\exists k.\, p(x_{i+k}) \wedge \neg r(x_{i+k}) \wedge (\forall j.\, j < k \Rightarrow \neg r(x_{i+j}))) \\ \vee \forall l.\, \neg r(x_{i+l}))\end{aligned}$$

Suppose we have x and i such that $\mathsf{trace\_q}\, q\, (\mathcal{T}_1 \| \mathcal{T}_2)\, x$ and $q(x_i)$ and $\neg r(x_i)$. Notice that we immediately have $\mathsf{trace\_q}\, q\, (\mathcal{T}_1 \| \mathcal{T}_2)\, x^i$.

We wish to prove by *reductio ad absurdum*, *i.e.* we assume the negation of the conclusion and we try to establish a contradiction. Assuming the negation of the conclusion give us the following extra assumptions: $\forall k.\, p(x_{i+k}) \Rightarrow r(x_{i+k}) \vee (\exists j.\, j < k \wedge r(x_{i+j}))$ and $\exists l.\, r(x_{i+l})$.

Let l be given such that r is true in the $i + l^{th}$ state of x. We know that there must be a smallest n such that r is true, *i.e.* $r(x_{i+n})$ and $\forall j.\, r(x_{i+j}) \Rightarrow n \leq j$. Using the assumptions we can easily derive the following property.

$$\forall j.\, j < n \Rightarrow \neg p(x_{i+j}) \tag{3.2}$$

Next we apply the slicing assumption (Ass 6) to the first hypothesis of the rule which gives us

$$(\mathsf{slice}\, \mathcal{T}_1\, (\mathsf{dep}_{p,r}\, \mathcal{T}_1)),\, q_{|(\mathsf{dep}_{p,r}\, \mathcal{T}_1)} \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r \tag{3.3}$$

In order to use this result we show that – given the different hypotheses – if x^i is a q-trace of the composed LTS and n is the smallest number such that $r(x_{i+n})$ holds, then slicing x^i with respect to $\mathsf{dep}_{p,r}\,\mathcal{T}_1$ returns an initial trace fragment of $\mathsf{slice}\,\mathcal{T}_1\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)$. This initial trace fragment takes at least $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n$ steps of the sliced LTS. Formally

$$
\begin{array}{l}
\mathcal{T}_1, q \models \mathsf{Exists}\ p\ \mathsf{Between}\ q\ r \wedge \\
\quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,r}\,\mathcal{T}_1) \mid p \wedge \\
\quad \mathsf{trace\_q}\ q\ (\mathcal{T}_1 \| \mathcal{T}_2)\ x^i \wedge \\
\quad r(x_{i+n}) \wedge (\forall j.\ r(x_{i+j}) \Rightarrow n \leq j) \Rightarrow \mathsf{trace\_q\_upto}\ \ q_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{slice}\,\mathcal{T}_1\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{slice\_trace}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n
\end{array}
$$

Using Lemma 1, we know that this initial trace fragment can be extended to a trace of $\mathsf{slice}\,\mathcal{T}_1\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)$. Let the extended trace starting with $\mathsf{slice\_trace}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)$ be called y. We can derive immediately that $q_{|\mathsf{dep}_{p,r}\,\mathcal{T}}(y_0)$. By using (3.3) we can immediately conclude that y satisfies the temporal property $\mathsf{Exists}\ p\ \mathsf{Between}\ q\ r$. If we use the mapping into LTL and the LTL semantics, we can instantiate the resulting formula with $i = 0$. This tells us that we have the following:

$$
\begin{array}{r}
(\exists k.\ p_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k) \wedge \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k) \wedge (\forall j.\ j < k \Rightarrow \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_j))) \\
\vee\ (\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_l))
\end{array}
$$

Notice that by using assumption (Ass 5) and the definition of V-equality and sliced states, we get for all k and l the properties

$$
\begin{array}{lcl}
(x_k)_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)} = y_l & \Rightarrow & p(x_k) \Leftrightarrow p_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_l) \\
(x_k)_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)} = y_l & \Rightarrow & r(x_k) \Leftrightarrow r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_l)
\end{array}
$$

We use these properties and a case distinction on the disjunction above to finish the proof.

Case 1 first disjunct Suppose we have a k such that $p_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k)$ and $\neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k)$ and $\forall j.\ j < k \Rightarrow \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_j)$. We make a case distinction on whether $\mathsf{nrss}V\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)$ is less than k.

Case 1.1 $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n < k$ In this case r does not hold in $y_{\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n}$. However by Lemma 4, this state is equivalent to $(x_{i+n})_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}$ and we already know that $r(x_{i+n})$, which gives us a contradiction.

Case 1.2 $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n \geq k$ By using Lemma 5 we know that we can relate y_k to a state x_m such that either $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,m = k$ or $\forall j.\ \mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,j < k$. The latter inequality gives us a contradiction with $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n \geq k$, therefore $\mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,m = k$. Next we make a case distinction on whether m is smaller than n.

Case 1.2.1 $m < n$ We know by (3.2) that $\neg p(x_{i+m})$, thus $\neg p_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k)$ using Lemma 4. This creates a contradiction with the assumption $p_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k)$.

Case 1.2.2 $n \leq m$ By monotonicity of the function nr_sliced_states (Lemma 2), we can derive $\mathsf{nr\_sliced\_states}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,n = \mathsf{nrss}\,(\mathsf{dep}_{p,r}\,\mathcal{T}_1)\,(x^i)\,m$, thus $x_{i+n} =_{(\mathsf{dep}_{p,r}\,\mathcal{T}_1)} x_{i+m}$ by Lemma 3. Therefore $r(x_{i+m})$ and thus $r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_k)$ by Lemma 4, which also leads to a contradiction.

Case 2 second disjunct First suppose that $\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_l)$. We had assumed that $r(x_{i+l})$. There exists a state in y that is the restriction of x_{i+l} to $\mathsf{dep}_{p,r}\,\mathcal{T}_1$. By the property above $r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}$ holds in this state, which gives us a contradiction with $\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\,\mathcal{T}_1)}(y_l)$.

3.6 Example

To illustrate how our factorisation method works in practice, consider the code fragments in Figures 3.5 and 3.6 (adapted from [41]). These fragments define a class `Buffer` which is accessed by 3 different threads: a consumer thread $\mathcal{C}$, a producer thread $\mathcal{P}$, and a thread $\mathcal{PB}$ that processes the buffer by moving the items in the incoming buffer to the outgoing buffer.

Typical properties that one may wish to verify include: if the incoming buffer is full it eventually will become non-full and; after the incoming buffer has become non-empty, eventually the outgoing buffer will also become non-empty. Using the specification patterns we can specify these properties formally as follows:

$$(\phi) \quad \texttt{!Buffer.inIsFull()}\ \mathsf{RespondsTo}\ \texttt{Buffer.inIsfull()}\ \mathsf{Globally}$$
$$(\psi) \quad \mathsf{Exists}\ \texttt{!Buffer.outIsEmpty()}\ \mathsf{After}\ \texttt{!Buffer.inIsEmpty()}$$

Using our factorisation rules we can show that it is sufficient to prove that these properties are guaranteed by the ProcessBuffer thread $\mathcal{PB}$ only. Notice that we only have to show the property ψ for traces where initially the incoming buffer is not empty. We can use any existing techniques for the verification of (multi-threaded) programs to establish this.

$$\begin{array}{r} \mathcal{PB} \models \phi \\ \mathcal{PB},\ \texttt{!Buffer.inIsEmpty()} \models \psi \end{array} \tag{3.4}$$

In addition, we need to show that the producer $\mathcal{P}$ and consumer $\mathcal{C}$ do not disturb the validity of the properties provided the appropriate conditions hold, *i.e.* the incoming buffer is full or the outgoing buffer is empty, respectively. In order to do this we first need to determine the appropriate dependency sets. Using a standard dependency analysis – as presented in [114] – we find the following:

$$\begin{array}{rcl} \mathsf{dep}_\phi\,\mathcal{PB} & = & \{\texttt{inhead}, \texttt{intail}, \texttt{inbound}\} \\ \mathsf{dep}_\psi\,\mathcal{PB} & = & \{\texttt{outhead}, \texttt{outtail}, \texttt{outbound}\} \end{array}$$

Now by appropriately instantiating the factorisation rules for RespondsTo Globally and Exists After, we find that we have the following extra proof obligations:

$$\begin{array}{l} \texttt{Buffer.inIsFull()} \models \mathcal{C}\|\mathcal{P}\ \mathsf{preserves}\ (\mathsf{dep}_\phi\,\mathcal{PB}) \mid \texttt{!Buffer.inIsFull()} \\ \texttt{Buffer.outIsEmpty()} \models \mathcal{C}\|\mathcal{P}\ \mathsf{preserves}\ (\mathsf{dep}_\psi\,\mathcal{PB}) \mid \texttt{!Buffer.outIsEmpty()} \end{array} \tag{3.5}$$

```
final class Buffer{
  int [] inbuf, outbuf;
  int inbound, outbound, inhead, outhead, intail, outtail;

public Buffer(int inb, int outb){
  inbound=inb; outbound=outb;
  inbuf=new int[inbound];
  outbuf=new int[outbound];
  inhead=0; outhead=0;
  intail=inbound-1; outtail=outbound-1;}

public synchronized boolean inIsFull(){
  return inhead==intail;}

public synchronized boolean outIsFull(){
  return outhead==outtail;}

public synchronized boolean inIsEmpty(){
  return inhead==((intail+1)%inbound);}

public synchronized boolean outIsEmpty(){
  return outhead==((outtail+1)%outbound);}

public synchronized void add(int o){
  while(inIsFull())
    try{wait();}catch(InterruptedException e){};
  inbuf[inhead]=o;
  inhead=(inhead+1)%inbound;
  notifyAll();}

public synchronized void process(){
  while(inIsEmpty())
    try{wait();}catch(InterruptedException e){};
  intail=(intail+1)%inbound;
  while(outIsFull())
    try{wait();}catch(InterruptedException e){};
  intail=(intail+1)%inbound;
  outbuf[outhead]=inbuf[intail];
  outhead=(outhead+1)%outbound;
  notifyAll();}
public synchronized int take(){
  while(outIsEmpty())
    try{wait();}catch(InterruptedException e){};
  outtail=(outtail+1)%outbound;
  notifyAll();
  return outbuf[outtail];}
```

Figure 3.5: Code fragment defining a class `Buffer`.

```
final class ProcessBuffer extends Thread{
  Buffer buf;
  public ProcessBuffer(Buffer b){buf=b;}
  public void run(){
    while(true)buf.process();}
}

final class Producer extends Thread{
  Buffer buf;
  public Producer(Buffer b){buf=b;}
  public void run(){
    int i=0;
    while(true){buf.add(i); i++;}}
}

final class Consumer extends Thread{
  Buffer buf;
  public Consumer(Buffer b){buf=b;}
  public void run(){
    while(true)System.out.println(buf.take());}
}
```

Figure 3.6: The buffer's three parallel threads: `Consumer`, `Producer` and `ProcessBuffer`.

These proof obligations are satisfied. We explicitly use the fact that the producer $\mathcal{P}$ does not produce any new elements when the incoming buffer is full and *vice versa:* that the consumer $\mathcal{C}$ does not take any elements when the outgoing buffer is empty. Note that without the extra conditions these factorisations would not have been possible. From (3.4) and (3.5) we can conclude the following:

$$\mathcal{PB} \| \mathcal{C} \| \mathcal{P} \models \phi$$
$$\mathcal{PB} \| \mathcal{C} \| \mathcal{P} \models \psi$$

Notice that if these three threads had been used in a larger context, many of the other typical properties of buffers (*e.g.* all elements that are taken first must have been added) can be factorised to these three threads only; for all other threads one only would have to show that they do not disturb the validity of the property.

It is worth using this buffer example to compare our approach to the three discussed in the introduction of this chapter: slicing, abstraction and atomicity checking. Recall that slicing techniques aid verification by removing instructions irrelevant to a property being checked. For our example, slicing techniques would be of no benefit since all three classes use the same variables. Abstraction techniques involve "abstracting" a program P into a smaller, more manageable transition system representing P. In our case this would not be particularly worthwhile, since the level of the property is close to the level of the application. Atomicity checking determines for each method whether the interleavings of its instructions gives the same result as executing its instructions without interleavings. Because we are more interested in verifying sequences of methods/actions, rather than conducting verification on a method *per* method basis, atomicity checking is not relevant in this case.

3.7 Proofs of lemmata

In this penultimate section we present proofs of the lemmata we have described throughout this chapter. We again refer the interested reader to `ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/Factorisation/` for all Isabelle/HOL proofs.

Lemma 1. $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j \Rightarrow \exists x'.\ \mathsf{trace\_pred}\ \mathcal{T}\ x' \wedge (\forall i.\ i < j \Rightarrow x_i = x'_i)$

Proof. To prove this lemma we first introduce the recursive function make_trace. This function builds an arbitrary trace for an LTS $\mathcal{T}$ given an initial state s. As long as a transition is enabled, then it makes an arbitrary transition, otherwise it stutters forever. (Note that *SOME* represents the Hilbert choice operator in Isabelle; it denotes the possible presence of a variable.) For a natural number i we define make_trace formally as follows:

$$\begin{array}{lcl}
\mathsf{make_trace}\ \mathcal{T}\ s\ 0 & = & s \\
\mathsf{make_trace}\ \mathcal{T}\ s\ (\mathit{Suc}\ i) & = & \mathsf{if}\ \exists a.\ \mathsf{enabled}\ \mathcal{T}\ (\mathsf{make_trace}\ \mathcal{T}\ s\ i)\ a \\
 & & \mathsf{then}\ \mathit{SOME}\, t.\ \exists a.\ (\mathsf{make_trace}\ \mathcal{T}\ s\ i) \xrightarrow{a} t \\
 & & \mathsf{else}\ \mathsf{make_trace}\ \mathcal{T}\ s\ i
\end{array}$$

We assume $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j$. In the LHS of the lemma, let x' be the trace whereby $x'_i = x_i$ if $i < j$ and $x'_i = \mathsf{make\_trace}\ \mathcal{T}\ x_j\ (i-j)$ otherwise. For readability we use y_i to denote $\mathsf{make\_trace}\ \mathcal{T}\ x_j\ (i-j)$. After simplifying, we find we need to prove the following two conjuncts using our assumption $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j$.

$$\begin{array}{l}
\mathit{Suc}\ i < j \Rightarrow\ (\exists a.\ \mathsf{enabled}\ \mathcal{T}\ x_i\ a \Rightarrow \exists a.\ x_i \xrightarrow{a} x_{\mathit{Suc}\ i} \\
\qquad \wedge \forall a.\ \neg\mathsf{enabled}\ \mathcal{T}\ x_i\ a \Rightarrow \forall k.\ (k < j \Rightarrow i \leq k \Rightarrow x_k = x_i) \wedge (j \leq k \Rightarrow y_k = x_i))
\end{array}$$

$$\begin{array}{l}
j \leq \mathit{Suc}\ i \Rightarrow\ (i < j \Rightarrow\ (\exists a.\ \mathsf{enabled}\ \mathcal{T}\ x_i\ a \Rightarrow \exists a.\ x_i \xrightarrow{a} x_j \\
\qquad\qquad \wedge \forall a.\ \neg\mathsf{enabled}\ \mathcal{T}\ x_i\ a \Rightarrow \forall k.\ (k < j \Rightarrow i \leq k \Rightarrow x_k = x_i) \\
\qquad\qquad\qquad\qquad \wedge (j \leq k \Rightarrow y_k = x_i))) \\
\qquad \wedge \\
\qquad (j \leq i \Rightarrow\ (\exists a.\ \mathsf{enabled}\ \mathcal{T}\ y_i\ a \Rightarrow \exists a.\ y_i \xrightarrow{a} y_{\mathit{Suc}\ i} \\
\qquad\qquad \wedge \forall a.\ \neg\mathsf{enabled}\ \mathcal{T}\ y_i\ a \Rightarrow \forall k.\ i \leq k \Rightarrow y_k = y_i))
\end{array}$$

Case 1 first conjunct We assume $\mathit{Suc}\ i < j$.

Case 1.1 first conjunct This is proven using the definition of $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j$.

Case 1.2 second conjunct We assume $\neg\mathsf{enabled}\ \mathcal{T}\ x_i\ a$ and deduce $\mathsf{stutters\_upto}\ x_i\ j$ using our assumption $\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j$. Hence if $k < j$ and $i \leq k$ then $x_k = x_i$. Next assume $j \leq k$. We know $x_j = x_i$, since $\mathsf{stutters\_upto}\ x_i\ j$. By the definition of enabled, we have $\forall a\, t.\ x_i \xrightarrow{a} t$. From here we can prove $\forall j.\ \mathsf{make\_trace}\ \mathcal{T}\ x_i\ j = x_i$ by induction on j. We know $k - j \geq 0$, hence $y_k = x_i$.

Case 2 second conjunct First suppose $j \leq \mathit{Suc}\ i$.

Case 2.1 first conjunct Assume $i < j$, therefore $j = \mathit{Suc}\ i$.

Case 2.1.1 first conjunct Proven using the definition of $\mathsf{trace\_pred\_up\_to}\ \mathcal{T}\ x\ j$.

Case 2.1.2 second conjunct Assume $\neg$enabled $\mathcal{T}\, x_i\, a$. If $k < j$ and $i \leq k$ then obviously $x_k = x_i$. Next, we know stutters_upto $x_i\, j$ using the definition of trace_pred_upto $\mathcal{T}\, x\, j$. Hence we may deduce $x_i = x_j$. Using reasoning similar to case 1.2 we can prove $\forall j.$ make_trace $\mathcal{T}\, x_i\, j = x_i$ and hence $y_k = x_i$ if $j \leq k$.

Case 2.2 second conjunct Assume $j \leq i$. Using our definition of make_trace it is not difficult to prove trace_pred $\mathcal{T}$ (make_trace $\mathcal{T}\, x_j$).

Case 2.2.1 first conjunct If enabled $\mathcal{T}\, y_i\, a$, then by the definition of trace_pred we can deduce $\exists a.\ y_i \xrightarrow{a} y_{Suc\, i}$.

Case 2.2.2 second conjunct Suppose $\neg$enabled $\mathcal{T}\, y_i\, a$. By using the definition of trace_pred we can show stutters y_i, hence $y_k = y_i$ if $i \leq k$.

Lemma 2. $i \leq j \Rightarrow$ nrss $V\, x\, i \leq$ nrss $V\, x\, j$

Proof. This is easily proved by induction on j and a case distinction $i \leq n$ and $n < i$.

Lemma 3. nrss $V\, x\, i =$ nrss $V\, x\, j \Rightarrow x_i =_V x_j$

Proof. In order to prove this lemma we first need to prove the following result.

$$\text{nrss}\, V\, x\, i = \text{nrss}\, V\, x\, (i+j) \Rightarrow x_i =_V x_{i+j} \tag{3.6}$$

This is done by induction on j, hence we have nrss $V\, x\, i =$ nr_sliced_states $V\, x\, (i+n) \Rightarrow x_i =_V x_{i+n}$. Next assume nrss $V\, x\, i =$ nrss $V\, x\, (i + Suc\, n)$. We want to show $x_i = x_{i+Suc\, n}$. We consider the following two cases.

Case 1 $x_{i+Suc\, n} =_V x_{i+n}$ Therefore nrss $V\, x\, (i + Suc\, n) =$ nrss $V\, x\, (i+n)$ by the definition of nr_sliced_states, hence $x_i =_V x_{i+n}$ by our assumption and induction step. Since V-equality is transitive we deduce $x_i =_V x_{i+Suc\, n}$.

Case 2 $x_{i+Suc\, n} \neq_V x_{i+n}$ We know nrss $V\, x\, (i + Suc\, n) >$ nrss $V\, x\, (i+n)$ by the definition of nr_sliced_states. We can show nrss $V\, x\, i \leq$ nr_sliced_states $V\, x\, (i+n)$ by Lemma 2. This creates a contradiction with our assumption and we are done.

To prove Lemma 3 suppose nrss $V\, x\, i =$ nrss $V\, x\, j$. Next consider the following two cases.

Case 1 $i < j$ Let $j = j - i$ in result (3.6). Hence $x_i =_V x_j$.

Case 2 $j \leq i$ Let $i = j$ and $j = i - j$ in result (3.6). Hence $x_i =_V x_j$ by symmetry of V-equality.

Lemma 4. (slice_trace $V\, x$)$_{\text{nrss}\, V\, x\, i} = (x_i)_{|V}$

Proof. To prove this lemma we proceed by induction on i. By our definition of slice_trace it is easy to prove (slice_trace $V\, x$)$_0 = (x_0)_{|V}$. Hence we have (slice_trace $V\, x$)$_{\text{nrss}\, V\, x\, n} = (x_n)_{|V}$ and we want to show (slice_trace $V\, x$)$_{\text{nrss}\, V\, x\, (Suc\, n)} = (x_{Suc\, n})_{|V}$. For brevity, we write $\gamma_n =$ nrss $V\, x\, n$. Hence if $x_n =_V x_{Suc\, n}$, then $\gamma_n = \gamma_{Suc\, n}$ and $(x_n)_{|V} = (x_{Suc\, n})_{|V}$ by a property of sliced states. Therefore we are left to consider the case where $x_n \neq_V x_{Suc\, n}$, hence when $\gamma_{Suc\, n} = Suc\, (\gamma_n)$. Our new goal becomes (slice_trace $V\, x$)$_{Suc\, (\gamma_n)} = (x_{Suc\, n})_{|V}$. Expanding the definition of slice_trace, we find we need to prove the two conjuncts given below.

$$\exists j.\ Suc\, (\gamma_n) \leq \gamma_j \Rightarrow (x_{\text{least}\, j.\ Suc\, (\gamma_n) = \gamma_j})_{|V} = (x_{Suc\, n})_{|V}$$
$$\forall j.\ \gamma_j < Suc\, (\gamma_n) \Rightarrow (x_{\text{least}\, j.\ \forall k.\ j \leq k \Rightarrow \gamma_j = \gamma_k})_{|V} = (x_{Suc\, n})_{|V}$$

Case 1 first conjunct Assume $Suc\,(\gamma_n) \leq \gamma_j$. Suppose $j < Suc\,n$ such that $Suc\,(\gamma_n) = \gamma_j$. Therefore $j \leq n$ and $\gamma_j \leq \gamma_n$ by Lemma 2, which creates a contradiction. Hence we deduce that the least j is such that $j = Suc\,n$.

Case 2 second conjunct Assume $\forall j.\ \gamma_j < Suc\,(\gamma_n)$. We instantiate j as $Suc\,n$ which immediately creates a contradiction.

Lemma 5. $\exists j.\ (\mathsf{slice\_trace}\,V\,x)_i = (x_j)_{|V} \wedge (\mathsf{nrss}\,V\,x\,j = i \vee \forall k.\ \mathsf{nrss}\,V\,x\,k < i)$

Proof. Again we write γ_j for $\mathsf{nrss}\,V\,x\,j$. By expanding the definition of slice_trace and simplifying, we find we need to prove the two following conjuncts.

$$\begin{array}{rcl} i \leq \gamma_j & \Rightarrow & \exists k.\ (x_{\mathsf{least}\,j.\ i=\gamma_j})_{|V} = (x_k)_{|V} \wedge (\gamma_k = i \vee (\forall l.\ \gamma_l < i)) \\ \forall j.\ \gamma_j < i & \Rightarrow & \exists k.\ (x_{\mathsf{least}\,j.\ \forall l.\ j \leq l \Rightarrow \gamma_j = \gamma_l})_{|V} = (x_k)_{|V} \wedge (\gamma_k = i \vee (\forall m.\ \gamma_m < i)) \end{array}$$

Case 1 first conjunct Instantiate k with least $j.\ i = \gamma_j$, hence we need only prove the second conjunct $\gamma_{\mathsf{least}\,j.\ i=\gamma_j} = i \vee (\forall l.\ \gamma_l < i)$. It is easy to prove $\gamma_{\mathsf{least}\,j.\ i=\gamma_j} = i$ from $i \leq \gamma_j$.

Case 2 second conjunct First assume $\forall j.\gamma_j < i$. Next instantiate k with least $j.\ \forall l.\ j \leq l \Rightarrow \gamma_j = \gamma_l$. Again we are left with the second conjunct. Instantiate j in our assumption with m and we are done.

3.8 Conclusions and future work

We have presented a method to factorise the verification of temporal properties for multi-threaded programs over their different threads. Contrary to other approaches that aim to reduce the verification burden by eliminating unnecessary verification tasks for the entire application, our approach is more modular in nature. We decompose the program into different parts for which different verification tasks exist, giving added flexibility to program verification. We feel that our technique can be used to improve the applicability of these other techniques.

As a property specification language we have used the specification patterns developed by the Bandera team. This language, along with our program model, has been formalised in Isabelle/HOL. We have designed 25 rules that describe the factorisation of a given temporal property and have proven each rule correct with respect to our formalisation. We also identified and corrected minor deficiencies within the patterns.

As future work we would like to develop an automatic technique to check for the preservation of variables. We believe that it will be possible to define this as an extension of existing techniques for checking so-called frame conditions [135; 28], *i.e.* specification clauses that describe which variables may be modified by a method.

A natural extension to our approach will be to take invariants into account (following [119]). If a property J is known to hold in all reachable program states, *i.e.* if it is an invariant, then this can be used to ease the verification process. The factorisation rules could be changed as follows:

$$\frac{\mathcal{T}_1, J \models \phi \qquad C_1, J \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{C_2}\ \mathcal{T}_1)\ |\ C_3}{\mathcal{T}_1 \| \mathcal{T}_2, J \models \phi}$$

Intuitively the proof rule would now read: in order to prove that the composed system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies property ϕ, assuming that we have an invariant J, it is sufficient to prove that (1) $\mathcal{T}_1$ satisfies ϕ assuming the invariant J, and (2) $\mathcal{T}_2$ preserves the variables on which V depends, also assuming the invariant J. One could also imagine using other, more elaborate properties as additional assumptions to the factorisation. It is the subject of future work to study those kind of properties which would be useful.

Chapter 4

Second-order principles in object-oriented specification languages

This chapter leaves the framework of Java program verification and instead looks at object-oriented program specification and verification in general. Within this setting, pointers and object references can be considered as relations between the elements of a data structure. Often when we specify properties of these data structures, we describe properties of relations. Hence it is important to be able to talk about relations and their properties when specifying object-oriented programs or programs with pointers. Many interesting properties of relations such as finiteness and generatedness – and operations on relations such as transitive closure – are not expressible in first-order logic (FOL). Hence neither are they expressible in first-order fragments of specification languages. In this chapter – which expands generally upon an existing paper written with Bernhard Beckert [14] – we give an overview of the different ways such properties and operations can be expressed in various logics, with particular emphasis on extensions of first-order logic, *i.e.* transitive closure logic, fixed point logic, and first-order dynamic logic. Within the chapter we also discuss which of these extensions already are – or in fact should be – implemented within specification languages. We feel that such a discussion is necessary since implementations of extensions of first-order logic within specification languages are often *ad hoc*.

4.1 Motivation

When it comes to specifying object-oriented programs, we need to be able to: (a) refer to a set of particular objects in an object structure; and (b) talk about the properties of the relation between the objects. As an example consider the definition of sets of related objects which are used in an assignable clause. (Recall from Chapter 2 that an assignable clause allows one to specify those parts of a program state that are exclusively allowed to change [108; 13].) To illustrate, suppose we have a linked list with objects of class `Node` having a `next` field. For a method say, `sortInPlace`, it would be useful to be able to write `list.next`* in the method's assignable clause, where * denotes some form of transitive closure. Its semantic intention would then be that the set of locations that are reachable from `list` using the field `next` may be modified during the method's execution. One may also wish to specify that the list is not cyclic; this requires special constructs not available in first-order logic.

We point out that most specification languages have some form of modification which allows them to extend beyond the limitations of first-order logic. For example the query language SQL implements fixed point logic, the Object Constraint Language OCL uses the `iterate` and `let` constructs, the Common Algebraic Specification Language CASL uses the notion of freeness, and the Java Modeling Language JML incorporates built-in recursion. However it is often the case that the modifications made to specification languages are done in a "make-do" fashion and their designers are unaware of the logic underpinning their decisions. This chapter attempts to clarify what is really going on within these specification languages.

Our work is carried out in the framework of the KeY project. Discussed briefly in Section 2.2.1, the KeY system is a commercial Computer Aided Software Engineering (CASE) tool augmented with specification and deductive verification functionalities [2; 91]. KeY uses the Unified Modeling Language UML for visual modelling of designs and specifications, along with OCL for specifying constraints and other expressions attached to the models [141]. The target language for program verification is Java. Both the specification language OCL and the verification language of the KeY tool – namely JavaCard DL, which is based on dynamic logic – have second-order elements. (These are described in Sections 4.3.4 and 4.4.4 respectively.) Our case study experience has shown that often there is a need for expressing second-order principles in a more useable and/or flexible way. In particular, an assignable clause has been recently implemented within KeY [13]. As the above example demonstrates, it would be advantageous to be able to express transitive closure in OCL in an easier fashion than the current method – which is by using the OCL `iterate` construct – described in Section 4.4.4.

The chapter is organised as follows: in Section 4.2 we look at how one goes about expressing properties of relations and composing relations. We discuss various properties which may or may not be expressed in first-order logic. This logic's lack of expressiveness leads us to an examination of a number of extensions of first-order logic in Section 4.3. In Section 4.4 we discuss several specification languages and the approaches they take in determining properties of relations. Finally, we draw conclusions in Section 4.5.

4.2 Relations and relational formulae in a FOL setting

We are interested in both expressing properties of relations and composing relations in relational formulae. In this section we provide the basic definitions for these notions and briefly discuss relational algebra. We conclude by describing a number of properties which can or cannot be expressed in first-order logic. However before we begin, we need to stipulate what we mean by a relation within an object-oriented language. Following [124] we say that a relation expresses (the symmetric form of) those associations which are represented in a programming language as pointers or object references. Hence we model both object references and pointers as first-order functions on objects.

A property P of a relation R is said to be expressible if there is a closed formula $\phi_P(R)$ such that for all models M R^M has property P if and only if $\phi_P(R)$ is true in M. Here R^M is the (single) interpretation of relation R in model M. The formula $\phi_P(R)$ must be effectively constructible from any given R in a uniform way. This notion is extended to properties of tuples of relations. Formally, a property is a relation on relations. A composition C of relations $R_1,\ldots,R_k$ is expressible if there is a formula $\psi_P(R_1,\ldots,R_k)(x,y)$ with two free variables x and y such that

$(\psi_P(R_1,\ldots,R_k))^M$ is the relation composed from $R_1^M,\ldots,R_k^M$. Here $(\psi_P(R_1,\ldots,R_k))^M$ is the (single) interpretation of $\psi_P(R_1,\ldots,R_k)(x,y)$ in model M. The formula $\psi_P(R_1,\ldots,R_k)$ must be effectively constructible from any given $R_1,\ldots,R_k$ in a uniform way. Formally, a composition is a function on relations. Note that the constructibility of ϕ and ψ neither implies the decidability of P, nor the computability of C. This is because the validity of the constructed formula is in general undecidable. Moreover the composition of relations may be iterated, but the properties themselves cannot be iterated.

Relational algebra is a formal system used for manipulating relations. The set of its operations may vary per definition, but it usually includes set operations – since relations are sets of tuples – and special operators defined for relations such as select, project and join. The select operator selects tuples from a relation whose attributes meet the selection criteria (which is normally expressed as a predicate). The project operator selects certain attributes from a single relation, discarding the rest. The join operator composes two relations. Relational algebra forms the basis of a multitude of relational query languages; these are used in order to manipulate the data of a relational database. We discuss aspects of one of the standard languages, SQL, in Section 4.4.2.

Examples of properties expressible in FOL We say that R is reflexive if $\forall x.xRx$ and R is transitive if $\forall x,y,z.(xRy \wedge yRz \rightarrow xRz)$. The concatenation of two relations R and S is expressible by $R \circ S \equiv \{(x,z) \mid \exists y.xRy \wedge ySz\}$. Note that we use the notation xRy for $(x,y) \in R$ and $R(x,y)$ respectively.

Examples of properties not expressible in FOL Properties that demand the finiteness of certain sets of elements are not expressible. For example "all elements are at most related to a finite number of other elements". Furthermore, many properties that demand the existence of a finite but unknown number of elements which are related in a certain way are not expressible. For example quantifications such as $\exists n.\exists x_1 \ldots x_n$ (which are routinely used in mathematical notation) do not exist in first-order logic and often cannot be expressed by any other means. Another typical but important example is transitive closure. The transitive closure of a relation R is the relation $TC(R)$ such that for all elements x and y, the relation $TC(R)(x,y)$ holds if and only if there is a finite number of intermediate points $z_0,\ldots,z_n$ where n is a natural number and $x = z_0$, $y = z_n$ and $z_{i-1}Rz_i$ for $1 \leq i \leq n$. Accordingly, one cannot express in first-order logic that some point b is R-reachable from some other point a, *i.e.* $TC(R)(a,b)$. An alternative – yet equivalent – definition of $TC(R)$ is one such that the following conditions hold: (1) $TC(R)$ is transitive; (2) $R \subseteq TC(R)$ and; (3) if R' is transitive and $R \subseteq R'$ then $TC(R) \subseteq R'$. (The third condition is not expressible in first-order logic since it implicitly quantifies over all R'.)

It is important to note that the transitive closure of a relation can be expressed in a first-order logic setting if the structure is both finite and acyclic. For example Baar shows in [7] that it is possible to define transitive closure within the specification language OCL without using the `iterate` construct. Building on a pre-existing translation of OCL into first-order logic, Baar defines a first-order representation of the transitive closure R^* of a relation R. Although counter models can be found whereby the interpretation of R^* is not the transitive closure of R, he finds that these all correspond to unlawful UML object diagrams which are infinite in scope or of a cyclic nature. By imposing a restriction to finite and acyclic models, R^* then becomes

a correct first-order definition of transitive closure. Hence it is possible to express transitive closure for finite and acyclic relations, but to express the finiteness of a relation we must extend first-order logic.

4.3 Extensions of FOL for expressing properties and compositions

In this section we present a number of extensions of first-order logic including transitive closure logic, fixed point logic and first-order dynamic logic. These extensions allow us to express various properties of relations that cannot be expressed in first-order logic alone.

4.3.1 Transitive closure logic

First-order logic extended by a transitive closure operator – written $FO(TC)$ and called transitive closure logic – was first introduced by Immerman [75]. If we let the formula $\phi(\bar{x},\bar{y})$ represent a binary relation on two n-tuples of domain variables – which range over the universe of a Kripke structure – then the reflexive transitive closure of this relation is expressed by $TC_{\bar{x},\bar{y}}\phi(\bar{x},\bar{y})$, or more succinctly $TC\phi$. Strict transitive closure is denoted $TC^s\phi$. This represents the transitive closure of ϕ as opposed to the reflexive transitive closure of ϕ. The restriction $FO^2(TC)$ is such that only two variables x and y may appear in a formula ϕ. As an example, the following formula expresses that "there is a path of a-edges from x to a vertex where p holds".

$$\exists z.\ ((TC_{x,y}R_a(x,y))(x,z) \wedge p(z))$$

Reachability logic $\mathcal{RL}$ is a fragment of $FO^2(TC)$ with an unbounded number of boolean variables in addition to the two domain variables x and y [4]. Boolean variables are first-order variables restricted to range over 0 and 1. Formulae of the logic are constructed using an adjacency formula $\delta(x,\bar{b},y,\bar{b'})$ which is a binary relation between two n-tuples $(x,b_1,\ldots,b_{n-1})$ and $(y,b'_1,\ldots,b'_{n-1})$. This is in fact a disjunction of conjunctions where each conjunction contains at least one of the following: $x = y$, $R_a(x,y)$, or $R_a(y,x)$ for some binary relation R_a. Hence the adjacency formula necessarily implies that there is an edge from x to y, or an edge from y to x, or that x is equal to y. Conjuncts may also contain expressions of the form $\neg(b_i = b_j)$, $b_i = 0$, or $b_i = 1$. For $\phi \in \mathcal{RL}$ the formulae $NEXT(\delta)\phi$, $REACH(\delta)\phi$ and $CYCLE(\delta)\phi$ are also formulae of $\mathcal{RL}$. They are given the following semantics.

$$\begin{array}{rcl} NEXT(\delta)\phi & \equiv & \exists y.\ (\delta(x,\bar{0},y,\bar{1}) \wedge \phi[y/x]) \\ REACH(\delta)\phi & \equiv & \exists y.\ (TC\delta)(\delta(x,\bar{0},y,\bar{1}) \wedge \phi[y/x]) \\ CYCLE(\delta) & \equiv & (TC^s\delta)(\delta(x,\bar{0},x,\bar{0})) \end{array}$$

Hence it is possible to describe in this logic: steps out of the current vertex x, paths out of x, and cycles from x back to itself.

Importantly, the boolean variables allow Propositional Dynamic Logic (PDL) and the variation of Computational Tree Logic, CTL* to be embedded in $\mathcal{RL}$. Consider the PDL formula $\langle\alpha\rangle p$, which is a true property of a state s whenever there is some state t in which p holds that is reachable from s by execution of α. The regular expression α can be translated into an non-deterministic finite automaton N_α with n states. Within the framework of $\mathcal{RL}$

the adjacency formula of α is a translation of the transition relation of N_α, whereby each state of the automaton is represented by $k \equiv 1 + \log n$ bits with $\bar{0}$ and $\bar{1}$ representing the initial and final states respectively. For example, if α is the sequential composition $\pi_0;\pi_1$ then a transition from state s to state t in $N_{\pi_0;\pi_1}$ is represented by the adjacency formula $R_{\pi_0}(x,y) \wedge b_1 \ldots b_k = s \vee R_{\pi_1}(x,y) \wedge b'_1 \ldots b'_k = t$ where $b_1 \ldots b_k$ is the initial state and $b'_1 \ldots b'_k$ is the final state. Hence an example of a formula in $\mathcal{RL}$ is $REACH(\delta)p$ where $\delta(x,b_1,b_2,y,b'_1,b'_2)$ is $(R_{\pi_0}(x,y) \wedge b_1 b_2 = 00 \wedge b'_1 b'_2 = 01) \vee (R_{\pi_1}(x,y) \wedge b_1 b_2 = 01 \wedge b'_1 b'_2 = 11)$. This has the meaning that it is possible to take the path of a π_0-edge followed by a π_1-edge to a point where p holds; this is just $\langle \pi_0;\pi_1 \rangle p$ in PDL.

4.3.2 Regular expressions over relations

Kleene algebras are algebraic structures that generalise the operations of regular expressions. A Kleene algebra consists of a set K with binary $+$ and $\cdot$ operations, a unary operation *, and constants 0 and 1. In general the algebra's operational semantics depends on the model, but typically * involves some notion of finite iteration. A Kleene algebra gives rise to a relational algebra extended with reflexive transitive closure when the following interpretations of the operations are made: operation $\cdot$ as join; element 0 as the null/empty relation; element 1 as the identity relation; and * as the reflexive transitive closure of a relation.

As we have mentioned previously, an extension of first-order logic which allows us to write `list.next`* – or more generally, allows us to use regular expressions in order to describe terms or sets of terms – would be very useful. There currently exists a number of first-order approaches which allow for an extended syntax for terms. For example, in [31] recursive term definitions are added to first-order logic.

Rather than use regular expressions and Kleene algebras to extend first-order logic, it is possible to manipulate first-order formulae such that they fulfil a purpose similar to that of regular expressions. Two ways to define words and/or formal languages are by using: (1) predicate logics, such that each model corresponds to a word in the language; and (2) modal logics, such that each path in a Kripke structure corresponds to a word. There is a large amount of literature on the latter. For (1) we fix a family of signatures Σ_A. They contain the binary relation symbol $<$, a constant symbol first, a unary postfix function $+1$, and for every a in the alphabet A, we have the unary relation symbol Q_a. The set of words over A is denoted A^*. For $w \in A^* \backslash \{\Lambda\}$, where Λ is the empty word, the associated Σ_A-structure is denoted $\mathcal{M}_w$ (the empty model is not possible). The formula $\mathcal{M}_w \models Q_a(0)$ holds true if and only if the first letter of w is a. The formula $\mathcal{M}_w \models Q_b(1)$ holds true if and only if the second letter of w is b, *etc.* For (2) we express information about semi-structured data – represented as a graph – by imposing constraints on the possible paths through the graph. Such a constraint might be "all objects reachable by a path p are also reachable *via* a path q", where p and q are sequences of labels possibly involving regular expressions. In order to check that the constraints hold, we re-cast them as model or satisfiability checking tasks in some logic (usually modal). For example, see [3] where this is done using propositional dynamic logic, and [42] where this is done using monadic second-order logic.

4.3.3 Fixed point logic

Fixed point logics are particularly well-suited for modelling recursion and have consequently found applications in various areas of computer science such as database theory, finite model theory and formal verification. Following [95; 44], for a set A and a function $F : \wp(A) \to \wp(A)$, a fixed point P of F is any set $P \subseteq A$ such that $F(P) = P$. A fixed point Q is called the least (greatest) fixed point of F if and only if $Q \subseteq P$ ($P \subseteq Q$) holds for all fixed points P of F. The function F is said to be monotone if $F(X) \subseteq F(Y)$ for all $X \subseteq Y \subseteq A$. A well-known theorem by Knaster and Tarski states that every monotone function has a least and a greatest fixed point [137]. Below we use ordinals to define the inductive fixed point of a function F. Ordinals are numbers used to denote the position in an ordered sequence. Ordinals which do not have an immediate predecessor are called limit ordinals, *e.g.* for any limit ordinal λ and ordinal α, $\alpha < \lambda$ implies $\alpha + 1 < \lambda$. For limit ordinals λ and the monotone function F, consider the sequence $(X^\alpha)_{\alpha \in Ord}$ of sets $X^\alpha \subseteq A$ defined as follows:

$$\begin{aligned} X^0 &= \emptyset \\ X^{\alpha+1} &= F(X^\alpha) \\ X^\lambda &= \bigcup_{\xi<\lambda} X^\xi \end{aligned}$$

A fixed point X^∞ is reached in this sequence whereby $X^\infty = X^\alpha$ for the least ordinal α such that $X^\alpha = X^{\alpha+1}$. This fixed point X^∞ is called the inductive fixed point of F. A second theorem by Knaster and Tarski states that the least and inductive fixed points coincide, hence any least fixed point of a monotone function can be defined inductively by a sequence of sets as described above. Dually, the greatest fixed point of a monotone function F can be defined inductively by the sequence $(X^\alpha)_{\alpha \in Ord}$ of sets $X^\alpha \subseteq A$ defined as follows:

$$\begin{aligned} X^0 &= A \\ X^{\alpha+1} &= F(X^\alpha) \\ X^\lambda &= \bigcap_{\xi<\lambda} X^\xi \end{aligned}$$

Note that if F is inflationary rather than monotone, *i.e.* $X \subseteq F(X)$ for all $X \subseteq A$, then X^∞ is called the inflationary fixed point of F. Next let τ be a signature, *i.e.* a finite set of relation symbols, and let $\mathcal{A}$ be a structure consisting of a universe A and interpretations for each relation symbol in τ. Consider a first-order formula $\phi(R,\bar{x})$ with R a free k-ary relation symbol not occurring in τ and $\bar{x}$ a k-tuple of free variables. On $\mathcal{A}$ the formula ϕ induces a fixed point operator $F_\phi : \wp(A^k) \to \wp(A^k)$ such that $F_\phi(R) = \{\bar{a} \mid (\mathcal{A},R) \models \phi(\bar{a})\}$. Here $(\mathcal{A},R) \models \phi(\bar{a})$ means that formula ϕ is satisfied by the interpretation that assigns to each variable x_i of $\bar{x}$ the element a_i of $\bar{a} \in A^k$.

Below we investigate three fundamental fixed point logics: monotone, least and inflationary. We begin by discussing monotone fixed point logic. Using this logic we can nest inductive definitions; from one fixed point built-up from a formula we can define another.

Monotone fixed point logic This logic is the extension of first-order logic by the following rule: if R is a free k-ary relation variable, $\bar{x}$ is a k-tuple of free first-order variables, $\bar{t}$ is a k-tuple of terms and $\phi(R,\bar{x})$ is a formula such that the corresponding operator F_ϕ is monotone on all

structures, then $[lfp_{R,\bar{x}}\phi](\bar{t})$ is also a formula. For any structure $\mathcal{A}$ that provides an interpretation of the free variables of ϕ except for $\bar{x}$, $\mathcal{A} \models [lfp_{R,\bar{x}}\phi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the least fixed point of the operator defined by $\phi(R,\bar{x})$. As we have mentioned previously, the least and greatest fixed point of any monotone operator always exists. However it is undecidable as to whether a formula induces a monotone operator. In order to guarantee monotonicity on the operator one can restrict the formulae such that they are positive in the relation variable R. This leads us to the definition of least fixed point logic.

Least fixed point logic This logic is the extension of first-order logic by the following rule: if R is a free k-ary relation variable, $\bar{x}$ is a k-tuple of free first-order variables, $\bar{t}$ is a k-tuple of terms and $\phi(R,\bar{x})$ is a formula in which R occurs only positively, then $[lfp_{R,\bar{x}}\phi](\bar{t})$ is also a formula. For any structure $\mathcal{A}$ that provides an interpretation of the free variables of ϕ except for $\bar{x}$, $\mathcal{A} \models [lfp_{R,\bar{x}}\phi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the least fixed point of the operator defined by $\phi(R,\bar{x})$. Consider, for example, the directed graph (V,E) where V is a set of n vertices and $E \subseteq V \times V$ is a set of ordered pairs, *i.e.* edges. Then the transitive closure of E is defined as $[lfp_{R,x,y}(xEy \vee \exists z.\ (xRz \wedge zRy))](x,y)$.

Inflationary fixed point logic This logic can be considered the simplest non-monotone fixed point logic. It is the extension of first-order logic by the following rule: if R is a free k-ary relation variable, $\bar{x}$ is a k-tuple of free first order variables, $\bar{t}$ is a k-tuple of terms and $\phi(R,\bar{x})$ is a formula, then $[ifp_{R,\bar{x}}\phi](\bar{t})$ is also a formula. Let $\mathcal{A}$ be a structure which provides an interpretation of the free variables of ϕ except for $\bar{x}$. The operator $I_\phi(R) = \{\bar{a} \mid \bar{a} \in R \text{ or } (\mathcal{A},R) \models \phi(\bar{a})\}$ is inflationary and therefore has an inflationary fixed point R^∞. Hence $\mathcal{A} \models [ifp_{R,\bar{x}}\phi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the inflationary fixed point. An interesting result is that least and inflationary fixed point logics are equally expressive on arbitrary structures [94].

4.3.4 First-order dynamic logic

The principle of Dynamic Logic, DL, is to formulate statements about program behaviour by integrating programs and formulae within a single language [62; 92]. By permitting arbitrary programs α as actions of a labelled multi-modal logic, dynamic logic provides formulae of the form $[\alpha]\phi$ and $\langle\alpha\rangle\phi$. If during program execution we consider states as worlds of modal logic, then $[\alpha]\phi$ expresses that all (terminating) executions of the program α lead to states in which ϕ holds. Moreover $\langle\alpha\rangle\phi$ is a true property of a state s whenever there is some state t in which ϕ holds that is reachable from s by execution of program α. A Hoare-style specification $\{\phi\}\alpha\{\psi\}$ of partial correctness can be expressed as $\phi \rightarrow [\alpha]\psi$. In contrast to Hoare logic and temporal logic approaches to program verification, dynamic logic permits the expression of structural relationships between different programs by using multiple modalities. For example relative correctness statements such as $\langle\alpha\rangle\phi \rightarrow \langle\alpha'\rangle\phi$ are possible.

Provided that they are computable, dynamic logic can express properties of relations that are ordinarily not expressible in first-order logic. For example to express that y is reachable from x *via* applications of the function *next* we write: $\langle \text{while } (x \neq y) \text{ do } x := next(x)\rangle true$.

4.4 Specification languages

In this section we look at the approaches that specification languages take in defining transitive closure and similar properties of relations. Most require "hacks" to force a model's finiteness and acyclicity before transitive closure can be determined. An interesting and unique approach – described in Section 4.4.5 – is taken by the Java Modeling Language JML.

4.4.1 Alloy

The Alloy Analyzer implements an automatic analysis method for the formulae of relational logic [77; 78]. This logic acts as an intermediate language for the object modelling notation Alloy. It is a first-order logic with sets and relations whereby each formula is accompanied by a declaration that associates variables to their types. The combination of formula and declaration is called a problem. There are three kinds of type: set, relation, and function. Scalar variables are treated as singleton sets and sets are encoded as relations. For example a scalar variable v of set type T can also be represented as the relational type $T \rightarrow \mathit{Unit}$, where *Unit* is a special type designed for this purpose.

A "navigation" expression $s.r$ denotes the image of a set s under a relation r. The encoding of sets as relations allows a uniform syntax to be given to such expressions, *i.e.* if p represents a person, then *p.mother* will denote p's mother, whereas *p.parents* will denote the set of p's parents. Of particular interest to us is Alloy's transitive closure operator $+$. As an example, the relational logic formula $(p+) \cap \mathit{Id} = 0$ expresses that p is acyclic. Here *Id* is the identity relation and 0 is the empty relation. Those environments for which a relational logic formula is true are called models of the formula. To determine for a given formula whether a model exists (within a particular scope) the Alloy Analyzer places restrictions on the size of the sets of the basic types. A model is said to be within a scope of k if it assigns to each type a set consisting of no more than k elements.

4.4.2 SQL

In order to manipulate the data of a relational database, relational query languages – based on relational algebra – are used. The database query language SQL was adopted as an industry standard in 1986 [136]. Having undergone two major revisions SQL3 is now the current version. Unlike its predecessors SQL3 supports linear recursion; writing a recursive query involves writing the query expression r that you intend to apply the recursion upon, giving it a name R and then using that name in an associated query expression Q.

```
WITH
RECURSIVE R AS r
Q;
```

If we consider a query as a function on tables, then a recursive query computes the "fixed point table" [148]. Essentially, we start with R as an empty table. We then evaluate r using the (temporary) contents of R and replace R with this new value. As long as $R^{new} \neq R$ we continue to evaluate r and replace R by its new value. Once $R^{new} = R$ we compute Q using

the current contents of R and output the result. The following example outlines how we find Mary's ancestors from the schema `ParentChild(parent, child)`.

```
WITH
RECURSIVE AncestorDescendant(ancestor, descendant) AS
    ((SELECT * FROM ParentChild)
    UNION
    (SELECT ad1.ancestor, ad2.descendant
    FROM AncestorDescendant ad1, AncestorDescendant ad2
    WHERE ad1.descendant = ad2.ancestor))
SELECT ancestor FROM AncestorDescendant WHERE descendant = "Mary";
```

The first part of the above recursive definition – utilising * – is the base case. Its meaning is that all `parent-child` pairs are also `ancestor-descendant` pairs. Although initially we know nothing about ancestor-descendant relationships, after the first round we deduce that parents are ancestors and children are descendants. In each subsequent round we use the facts deduced in previous rounds to get more ancestor-descendant relationships. We eventually stop when no new facts can be proven.

When the query Q is non-monotone, *i.e.* by adding tuples to R we might cause some tuple to be removed from the result of Q, then the fixed point iteration may not converge. A way to circumvent this is to construct a dependency graph whereby: (1) each table R_i is a node; (2) there is a directed arc from R_i to R_j if R_i is defined in terms of R_j; and (3) the arc is labelled "-" if the query defining R_i is non-monotone with respect to R_j. The maximum number of - arcs on any path from R in the dependency graph is called the stratum of node R. A recursive query statement is said to be stratified if every node has a finite stratum, *i.e.* there are no cycles containing - arcs. Hence legal SQL3 recursive queries are required to be stratified. Note that in other languages with fixed point definitions, this technique can be used to exclude non-monotonicity cases that lead to fixed points being undefined.

4.4.3 CASL

The Common Algebraic Specification Language, CASL, has been developed by CoFI, the international Common Framework Initiative for algebraic specification and development [37]. The algebraic approach to software specification was conceived in the early 1970s, (see for example [149]). Programs are considered as algebras consisting of datatypes and operations; the intended behaviour of a program is specified by formulae involving these operations. The development of dozens of languages – all with slight variations in syntax and semantics – demanded the need for a common framework, hence CoFI was formed. The resulting specification language CASL features partial functions, subsorts, sort generation constraints, first-order logic, and structural and architectural specifications [107].

In CASL datatypes are specified using the keyword **type** and are given in terms of sorts (*i.e.* the types of values) and constructors. Datatypes may be declared as either **generated** or **free**. When a **generated** datatype is declared, then the corresponding sort is constrained to be generated only by the declared constructors. For example in the following specification of GENERATED_CONTAINER taken from the CASL User Manual [18], the generatedness constraint

is such that any value of sort *Container* is denoted by a term built only with operators *empty*, *insert* and variables of sort *Elem*.

spec GENERATED_CONTAINER [**sort** *Elem*] =
generated type $Container ::= empty \mid insert(Elem;\ Container)$
pred $\_\_is\_in\_\_ : Elem \times Container$
$\forall e, e' : Elem;\ C : Container$
$\bullet \neg(e\ is\_in\ empty)$
$\bullet\ e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
end

Note that within this specification the pairs of underscores "__" indicate place-holders for the binary predicate *is_in*, and the bullet-pointed list features "axioms" which constrain the predicate. Essentially, the generatedness constraint allows one to prove (by induction on the declared constructors) properties of values of the sort *Container*. A **free** datatype declaration has the same interpretation as the **generated** datatype declaration with the additional property that all distinct constructor terms of the same sort denote distinct values.

In CASL a "freeness" constraint – using the keyword **free** – can be imposed on a predicate declaration. This has an effect such that a predicate which is consistent with the given axioms, but not a consequence of the axioms, will be false *i.e.* predicates hold minimally. We can see this in the following specification (also taken from [18]). Here the transitive closure of a binary relation R on some sort *Elem* is specified.

spec TRANSITIVE_CLOSURE [**sort** *Elem* **pred** $\_R\_ : Elem \times Elem$] =
free { **pred** $\_R^+\_ : Elem \times Elem$
$\forall x, y, z : Elem$
$\bullet\ xRy \rightarrow xR^+y$
$\bullet\ xR^+y \wedge yR^+z \rightarrow xR^+z$ }
end

Since predicates hold minimally in models of free specifications, R^+ is actually the smallest transitive relation involving R.

4.4.4 OCL

The Object Constraint Language, OCL [110], is a sublanguage of the Unified Modeling Language, UML [56; 141]. Currently the industry standard, UML allows software developers to graphically specify, visualise and document models of software systems. OCL can be used to augment UML object models with additional textual information which cannot otherwise be expressed by UML diagrams. This additional information takes the form of side-effect-free expressions and constraints. An expression is a specification of a value. A constraint is a restriction of one or more values in (part of) the object-oriented model. The semantics of OCL constraints is defined by an evaluation function which maps – in a given object diagram – any constraint to one of the logical constants `true`, `false`, and `undefined`. Admissible diagrams are those whereby all constraints of the corresponding class diagram evaluate to `true`.

The type of an OCL expression is either predefined (`Boolean`, `Integer`, *etc.*) or it is the type of a class in the corresponding class diagram. Dot notation is used for accessing the attributes of objects. The basic data structures of OCL are the collections `Set`, `Bag` and `Sequence`. A `Bag` is a multiset, with possible repeated elements. A `Sequence` denotes ordered bags. Each of these data structures is parametric; we write `Set(T)`, `Bag(T)` and `Sequence(T)` for type `T`.

OCL does not have a primitive operator for transitive closure, but it does allow recursion. Consider the following OCL invariant in the context `Person`, where `ancestors` are recursively defined in order to represent the transitive closure of the relation defined by `parents`.

```
Person inv
  ancestors = parents -> union(parents.ancestors)
```

Note that both `ancestors` and `parents` are of type `Set(Person)`. The expression `parents.ancestors` computes the set of all ancestors of a set of parents and returns a value of type `Set(Person)`. Now suppose `A` is a parent of `B`, who in turn is a parent of `C`. Then the minimal object structure which solves the constraint is such that the parent of `B` is `A` and the ancestors of `C` include both `B` and `A`. However, additional solutions involve situations where `B` and `A` are both ancestors of each other and themselves. In our case we would prefer to use the minimal solution (corresponding to the least fixed point), but this cannot always be found: there may be more than one equivalent solution, or it may not even exist. A suggestion to uniquely characterise the minimal solution by mimicking induction over a natural number `n` is given in [39]. This is exhibited in the following OCL specification.

```
Person
  ancestors_up_to(n) = if (n==1) then parents
                       else parents -> union(parents.ancestors_up_to(n-1))
  Nat -> forall(n | ancestors_up_to(n) = ancestors_up_to(n+1)
                    implies ancestors = ancestors_up_to(n))
```

Of course this makes the assumption that the models are finite. Alternatively, as done in [30], we can use the OCL `let` construct to stipulate that the ancestor relationship must be acyclic. Note that `self` refers to any instance of the class in which it is specified.

```
Person inv
  let parents = self.parents
  let ancestors = self.parents -> union(self.parents.ancestors)
  in <some_expression_using_definition_of_ancestor>
```

The `let` construct is a new addition to OCL, introduced in version 2.0. The expression `let x = e1 in e2` evaluates expression e_2 with each occurrence of `x` replaced by the value of e_1. Its use avoids evaluating the same expression multiple times. However the construct's semantics within OCL is not entirely clear [30]. Whether arbitrary recursively defined expressions are allowed is uncertain. Therefore using `let` to define transitive closure is not advised.

In [101] the transitive closure of a relation is computed by coding the well known War-

shall's algorithm in OCL. This coding makes use of the OCL `iterate` construct which iterates through all items of a collection, verifying a given condition and possibly updating the value of a variable returned at the end of the iteration. The algorithm itself calculates the transitive closure of a directed graph (V,E) where V is a set of n vertices and $E \subseteq V \times V$ is a set of ordered pairs, *i.e.* edges. A path from vertex v_0 to v_k is denoted $v_0 \xrightarrow{*} v_k$ and is a sequence of edges $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$. The intuition behind Warshall's algorithm is this: if the graph contains paths $v \xrightarrow{*} w$ and $w \xrightarrow{*} u$ whose intermediate vertices belong to the set S, then the graph also contains a path $v \xrightarrow{*} u$ such that the intermediate vertices belong to $S \cup \{w\}$. The algorithm iterates from 1 to n. At the k^{th} iteration it selects paths whose intermediate vertices come from $\{v_1,\ldots,v_{k-1}\}$. Unfortunately the resulting OCL code of this algorithm is about one and a half pages in length; it is neither intuitive nor easy to read, and furthermore it requires the directed graph to be finite.

A transitive closure construct for OCL is proposed by Schürr in [131]. This is based on features of the path expression sublanguage of PROGRES, a graph transformation language (similar to OCL). If we use our "ancestors" example to illustrate this approach, then `A` is an ancestor of `C` (`A` is said to "conform to" `C`) if either `A` and `C` are the same person, or if there is a person `B` such that `A` is an ancestor of `B` and `C` is a child of `A`. Schürr writes the constraint to determine the minimal solution as follows:

```
Person inv:
  self.conformsTo = self.ancestors.conformsTo
                      -> including(self) -> asSet
```

Here `conformsTo` is an abbreviation for the following expression.

```
Person -> Set(Person) = self.ancestors* -> asSet
```

The transitive closure operator * is implemented to keep track of already visited objects and therefore avoids any cylic problems. Schürr defines it as follows:

```
self.ancestors* = self.ancestorsClosure(self)
    self.ancestorsClosure(visitedObj) =
    let S:... = self.ancestors -> excludeAll(visitedObj) in
    S -> collect(ancestorsClosure(S -> union(visitedObj))) -> asSet
```

Note that this definition will suffer from the unclear semantics of the `let` construct.

As mentioned previously, Baar points out in [7] that it is possible to define the transitive closure of relations known to be finite and acyclic. He writes the definition of `ancestors` as follows, where for a given object `x`, $Par(x)$ is the translation of `x.parents` and $APar(x)$ is the translation of `x.ancestors`.

$$APar(x) = Par(x) \cup \{y \mid \exists z.\ z \in Par(x) \wedge y \in APar(z)\}$$

This is transformed into the following first-order logic formula, where the relation symbols r and r^* are substituted for Par and $APar$, with $r(x,y)$ meaning $y \in Par(x)$ and $r^*(x,y)$ meaning

$y \in APar(x)$.

$$r^*(x,y) \Leftrightarrow (r(x,y) \vee \exists z.\ r(x,z) \wedge r^*(z,y)) \tag{4.1}$$

The formula (4.1) is interpreted by the structure (U,R,R^*) where U is a universe of variables, and R and R^* are interpretations of the relations r and r^*, respectively. Counter models for this formula are presented whereby R^* does not coincide with the transitive closure of R. However if the model (U,R,R^*) is finite and the axiom $\neg r^*(x,x)$ holds – enforcing R^*s acyclicity – then R^* is a correct definition of transitive closure.

4.4.5 JML

As mentioned in Section 2.1, the Java Modeling Language is currently the academic community's standard specification language for Java. JML specifications are formulated by making use of (side-effect-free) boolean Java expressions; they are written as Java comments following the symbols `//@`. The original JML tool is a pre-compiler designed to translate specified programs into Java programs which explicitly monitor assertions at runtime. Specification violations that are found throw Java exceptions. Since JML's conception in 1998, several tools have been developed which use JML as an input specification language. For a more extensive overview of the language and its tools, see Sections 2.2 and 2.2.1 respectively.

When specifying transitive closure JML manages to avoid the whole issue of acyclicity by defining recursive datagroups [108]. These have been designed primarily with frame-condition issues in mind. To solve the information hiding problem (*i.e.* that protected or private fields of a class should remain hidden from their clients) the `represents` clause was introduced to JML, allowing one to specify the representation of concrete fields by particular abstract fields. Hence protected or private fields in an implementation can be changed without changing the specification visible to its clients. Unfortunately, the use of abstract fields generated problems with the `assignable` clause. (Recall that a method's `assignable` clause specifies those locations that are permitted to be changed by execution of the method.) This was fixed by a `depends` clause which relates those locations used to determine an abstract location's values.

A datagroup can be modelled by an abstract location whose value contains no information. By using a `depends` clause, a location can be declared to be in a datagroup, therefore membership in a datagroup allows the locations in the datagroup to be modified whenever the datagroup is mentioned in the `assignable` clause. The licence to modify a datagroup implies the licence to modify the members of the datagroup as defined by a downward closure rule [96]. For any set of datagroups S, the downward closure of this set is the smallest superset of S such that for any group G in the closure of S, all nested datagroup members of G also belong in the closure of S. For example, consider the following Java linked list with `Node` objects having `next` and `value` fields.

```
class Node{
    Integer value;
    Node next;
}
```

Datagroups `nodeValues`, `nodeLinks`, and `linksAndValues` are defined recursively.

```
//@ in nodeLinks, linksAndValues
//@ maps next.nodeValues \into nodeValues;
//@ maps next.nodeLinks \into nodeLinks;
//@ maps next.linksAndValues \into linksAndValues;
```

Hence the JML specification below says that all node objects reachable from `list` may be changed whenever `sortInPlace` is executed.

```
//@ assignable list.nodeLinks;
void sortInPlace(Node list);
```

Such specifications rely on a least fixed point semantics for recursive definitions built into JML. Gleaned from mailing list discussions, the developers of JML have considered introducing regular expressions, *i.e.* writing `list.next`* in order to specify the `JMLObjectSet` of all objects reachable from `list` using the field name `next`. However this proposal has been rejected since the consensus among JML users seems to be that datagroups are an adequate enough solution.

4.5 Conclusions and future work

Although important properties of relations – and operations on relations – are not expressible in classical first-order logic, it is possible to extend first-order logic (*e.g.* with fixed point and transitive closure operators) in order to describe them. We find that all specification languages feature modifications which allow them to extend beyond the limitations of first-order logic. For example SQL implements fixed point logic, OCL uses the `iterate` and `let` constructs, CASL implements the notion of freeness, whereas JML incorporates built-in recursion. However it is often the case that the designers of these specification languages are not really aware of the logic underpinning their modifications. This chapter has attempted to clarify what is really going on regarding these first-order extensions.

Generally we have found that once integers are "available" in a specification language, it is possible to define transitive closure and other properties of – or operations on – relations in the language. Otherwise such properties and operations may only be defined using finite relations (which is mostly adequate). In our opinion the most effective and least complicated solutions are the approaches taken by CASL and JML, whereby the notion of freeness or minimal fixed points are either explicitly or implicitly built into the language. It still seems desirable to add regular expressions to specification languages – this would make specifications more intuitively easier to understand – however it is not yet clear how this should be done; this is the subject of future work.

Chapter 5

Proving correctness of JavaCard DL taclets using Bali

The JavaCard DL sequent calculus captures the semantics of JavaCard, the subset of Java designed to run on smartcards. Lightweight, stand-alone tactics or "taclets" have recently been introduced in order to implement JavaCard DL within the KeY verification tool. This chapter discusses a case-study into proving taclets sound using the independent "Bali" formalism of Java in the theorem prover Isabelle/HOL. Rather than take a foundational approach, *i.e.* by proving all rules from a small primitive set of rules (usually done by embedding the calculus into a theorem prover), we instead translate each taclet and prove its soundness with respect to the Bali operational semantics. We examine the JavaCard DL and Bali approaches, prove three pivotal taclets sound, and argue whether the method is useful in proving the relative correctness of JavaCard DL programs overall. This chapter expands upon an existing paper written by the author [139]. As opposed to the paper, we delve more deeply into the Bali operational semantics and JavaCard DL calculus so that the chapter is self-contained. Soundness proofs of the Bali translations can be found at `http://users.rsise.anu.edu.au/~kerryt/`.

5.1 Motivation

In the previous chapters we have discussed how we can make specification languages more expressive and how we can lighten the verification burden. In this chapter we look at the correctness of the verification tool itself. No matter how expressive the specification language it utilises, no matter how efficiently it performs its verifications, a verification tool is worthless if it does not accurately model the target programming language. For a calculus implemented within a tool, if we cannot show the soundness of its rules with respect to the semantics of the target programming language, then we can conclude very little about the "correctness" of a program's verification.

In [12] Beckert, Giese *et al.* introduced the concept of taclets to the implementation of interactive theorem provers. Taclets are lightweight tactics, *i.e.* proof advancing routines, with a simple syntax and semantics. They contain such information as: the logical content of the rule to be applied, restrictions or "guards" on their applicability, and heuristic information on whether a rule is either applied automatically or interactively. The KeY verification tool is based on a set of taclets which implement the JavaCard DL sequent calculus [91; 11]. This

calculus has approximately 300 axiomatic rules which capture the semantics of JavaCard. Of particular concern to the KeY team is the correctness of these JavaCard DL taclets, since new taclets can be introduced relatively easily. The main focus of this chapter is (proving) the correctness of the assignment taclets modelling local variable, field and array assignments. Since assignments are the basic state-changing statements of Java(Card) these taclets are of crucial importance to the KeY implementation.

In his PhD thesis [111] von Oheimb described the formalism of a subset of JavaCard called Java^{light} in the theorem prover Isabelle/HOL [76]. He defined an abstract syntax and static semantics for the language, including a type system and well-formedness conditions, and provided an operational/evaluation semantics. Using the formalism, von Oheimb proved the soundness of Java^{light}'s type system, justifying the type safety claims of Java's designers. The operational semantics have since been expanded upon and developed as part of the ongoing Bali project at the Technical University of Munich [8]. Although still missing a number of important features of Java such as multi-threading, we believe that the Bali operational semantics provide the best setting for verifying formal correctness of JavaCard DL taclets.

Note that a Hoare-style calculus for Java^{light} has also been formalised by von Oheimb [112]. This calculus has been proven both sound and complete with respect to the Bali operational semantics. To avoid confusion, we point out that the terms "Bali" and "Bali calculus" used throughout this thesis refer to the operational semantics for Java and its corresponding Isabelle/HOL formalism, *not* the Hoare-style calculus.

Related work This case-study further broadens and deepens work initiated in [128] by Sasse. In this paper several KeY rules for Java conditional statements are translated into the Bali syntax and one rule is proven correct. Complementary to the work presented here is the approach in [23] where derived, non-axiomatic taclets for JavaCard DL are proven sound relative to the core set of JavaCard DL axioms. Given a taclet *tac*, a "meaning" formula $M(tac)$ is derived. This formula captures both the logical content and operational meaning of a taclet, and is supposed to be valid if and only if all possible applications of the taclet are correct. To show that a taclet is correct – or to derive a taclet from existing rules – it is sufficient to prove the validity of the corresponding skolemised meaning formula.

This chapter is outlined as follows. Section 5.2 describes JavaCard DL's syntax and semantics and also presents the assignment rules of this calculus. Section 5.3 provides basic background material on the theorem prover Isabelle/HOL. Section 5.4 introduces Bali. We describe its syntax and semantics and present the assignment evaluation rules. Sections 5.5, 5.6 and 5.7 show how we translate the KeY implementations of (respectively) the local variable, field and array assignment rules into Bali and prove them correct. Finally we draw conclusions and discuss future work in Section 5.8.

5.2 KeY and JavaCard DL

Discussed in Section 4.1, KeY is an augmented commercial CASE tool with specification and deductive verification functionalities [2; 91]. Developed at the University of Karlsruhe, KeY uses the Unified Modeling Language UML for visual modelling of designs and specifications

along with UML's inherent Object Constraint Language OCL, for specifying constraints and other expressions attached to the models [141; 56]. The JavaCard DL calculus is used as the logical basis of the KeY system's program verification component.

Beckert's JavaCard DL calculus was first introduced in [11]. Considered on its own, first-order dynamic logic has a modality $\langle p \rangle$ for every program p [92]. The modality $\langle p \rangle$ represents successor worlds – called states – which are reachable by running the program p. In JavaCard DL p is simply any sequence of legal JavaCard statements. Furthermore, since Java(Card) is deterministic, there is either only one state reachable (if p terminates) or no state reachable (if p does not terminate). The formula $\langle p \rangle \phi$ expresses that p terminates in a state where ϕ holds. A formula $\phi \rightarrow \langle p \rangle \psi$ expresses that for every state s satisfying precondition ϕ, a run of the program p starting in s terminates in a state at which postcondition ψ holds. This is comparable to the Hoare triple $\{\phi\} p \{\psi\}$ where ϕ and ψ are first-order formulae. However unlike Hoare logic, in dynamic logic the formulae ϕ and ψ may contain programs.

5.2.1 Syntax and semantics

In order to define JavaCard DL's syntax, we follow [11] by first specifying its types along with the variables from which (logical) terms are built. We next define both the program p (such that it is allowable in the modal operator) and the JavaCard DL formulae themselves. We then specify what it involves to "update" a term or formula; an important feature of JavaCard DL. Finally, we define the syntax of JavaCard DL sequents.

Types The set of types contains: the primitive types of JavaCard, *i.e.* `boolean`, `byte` and `short`; the predefined classes `Object` and `String`; the classes defined in the program context; an array type $T[]$ for each primitive type and each class; the type *Null*; and abstract types. Abstract types may only exist in the non-program parts of a JavaCard DL formula, however they can be used to describe the behaviour of programs and as abstractions of object structures.

Variables JavaCard DL implements two kinds of variables: program variables (to appear in typewriter font) and logical variables (to appear in italics). Program variables are Java(Card) local variables whose values can be changed by programs. Logical variables are assigned the same values in all states and unlike program variables, are quantified. Both kinds of variables may be used in program as well as non-program parts of JavaCard DL formulae.

Terms These are referred to as "logical" terms in order to distinguish them from JavaCard expressions. They are constructed from program variables, logical variables and the (correctly-typed) constant and function symbols of all types. In particular the set of logical terms contains: all JavaCard literals for primitive types; string literals; and the `null` object reference literal of type *Null*. Moreover, if o is a term denoting an object of class C and `a` is a field of C, then o.`a` is a term. The same construction is valid if o is a class name and `a` is a static field of that class. If a is an array type term and i is a term of type `byte`, then $a[i]$ is also a term.

Programs JavaCard DL programs are essentially executable JavaCard code with two possible additions: programs can contain a special construct for method invocations and they may

also contain logical terms. These additions do not feature in the input formulae: they only appear in proofs and arise from rule applications. When a method is invoked in Java(Card) the flow of control passes from the method call into the method implementation itself. A method completes execution and returns to the caller when one of the following occurs: a `return` statement is executed, the end of the method is reached, or an uncaught exception is thrown. In order to handle the `return` statement correctly, it is necessary to record the program variable or object field that the result is to be assigned to. Hence the statement `methodcall`(old,x) is introduced. The parameter *old* stores the value of the `this` pointer at the method call's inception, and x is the variable or object field to which the returned value is assigned.

Formulae Atomic formulae are built from logical terms, the predicate symbols of all types, and the following: the equality predicate $\doteq$, the unary "definedness" predicate *isdef* and the binary instance predicate *instanceof*. Complex formulae are built from atomic formulae using the logical connectives $\neg, \wedge, \vee, \rightarrow$ and quantifiers $\exists$ and $\forall$ (which are applicable only to logical variables). Not to be forgotten is the modal operator $\langle p \rangle$. If p is a program and ϕ is a formula, then $\langle p \rangle \phi$ is also a formula.

Updates These are used in JavaCard DL in order to handle aliasing. For example the different object references o_1 and o_2 may be aliases for the same object such that changing a field of o_1 changes the same field of o_2. A mechanism is needed whereby the field of o_2 is updated by the changes made to the same field of o_1. Hence updates of the form $\{x := t\}$ or $\{o.\mathtt{a} := t\}$ are attached to terms and formulae. Here x is a program variable, $o.\mathtt{a}$ is a term and t is a logical term of compatible type. Suppose $\mathcal{U}$ is an update, t is a term and ϕ is a formula, then $\mathcal{U}t$ and $\mathcal{U}\phi$ are also a term and formula respectively. The semantics of an update is such that the term or formula that it is attached to is to be evaluated after changing the state accordingly.

Sequents A sequent is of the form $\Gamma_1, \ldots \Gamma_n \vdash \Delta_1, \ldots \Delta_m$ where Γ_i and Δ_i are sets of JavaCard DL formulae and $m, n \geq 0$. Its semantics is such that the conjunction of the Γ_i's implies the disjunction of the Δ_i's.

The semantics of JavaCard, following [61; 19], is used to define the semantics of JavaCard DL. The models of JavaCard DL are Kripke structures; albeit we refer to states instead of worlds. All states of a model have the same universe containing a sufficient number of elements of each type. Moreover they contain an infinite number of classes, array types and the special value *null* which is the only element of type *Null*. In every state a different value of the appropriate type may be assigned to: program variables, including the `this` pointer; the fields of all objects, including arrays; and the static fields of all types. Furthermore, variables and fields of type T may be assigned a value of type T' if T' is a subtype of T, *i.e.* $T' \preceq T$. Since *Null* is a subtype of all object types, the value *null* may be assigned to any variable or field of an object type. It is important to note that states neither contain information about control flow, nor whether an exception has been thrown.

As alluded to previously, the semantics of a program is a state transition. A program p assigns to its initial state s the set of all states reachable by running p. Because of determinism, this set is either empty (if p does not terminate) or contains only one state (if p terminates).

Programs that terminate abruptly are considered to be non-terminating. The semantics of a logical term t within a program is the same as that of a side-effect free JavaCard expression whose evaluation gives the same value as t.

For a formula ϕ that does not contain a program, the notion of ϕ being satisfied by a state is defined as in first-order logic. A formula $\langle p \rangle \phi$ is satisfied by a state s if the program p, starting in s, terminates normally in a state in which ϕ is satisfied. Routinely, a formula is satisfied by a model M if it is satisfied by one of the states of M and is valid in M if it is satisfied by all states of M.

5.2.2 Aspects of the JavaCard DL calculus

In this section we outline some of the ideas behind the JavaCard DL calculus and discuss the assignment rule which is of particular relevance to us. We refer the reader to [11] for a more thorough presentation of the JavaCard DL rules. The sequent rules have a semantics such that if the premises are valid, then the conclusion is valid. The rules are applied from bottom to top, *i.e.* the proof search starts with the original proof obligation at the bottom. Essentially the rules perform a symbolic program execution whereby expressions and statements are reduced stepwise to side-effect free assignments.

JavaCard DL rules operate on the first "active" statement p of the program $\pi\, p\, \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces, labels, and the beginnings of `try_catch` blocks and/or method invocation blocks. The "rest" of the program ω contains the remaining active statements along with the matched endings of π.

The assignment rule is given below.

$$\frac{\Gamma \vdash \mathcal{U}\{loc := val\}\langle \pi\, \omega \rangle \phi}{\Gamma \vdash \mathcal{U}\langle \pi\, loc = val;\, \omega \rangle \phi}$$

Here $\mathcal{U}$ is an arbitrary sequence of updates, *val* is a logical term with no side effects and *loc* is one of the following: a program variable v, a field access $o.\mathtt{a}$, or an array access $a[i]$. Essentially the assignment rule just adds the assignment to the list of updates $\mathcal{U}$. Special simplification rules are used to compute the result of applying an update to logical terms and formulae not containing any programs. Without applying them, simplifications of updates may be performed at any time.

5.3 Isabelle basics

Isabelle [116] is a generic interactive theorem prover which allows the encoding of different object logics. Of particular relevance to us is Isabelle/HOL, which is Isabelle instantiated with Church's higher order logic (HOL) [33]. Its syntax is similar to ML and appears in a standard way, for example the arrow $\longrightarrow$ is the right-associative infix implication symbol. The long arrow $\Longrightarrow$ represents meta-implication and appears in rules or theorems of the form $[\![P_1; \ldots; P_n]\!] \Longrightarrow C$ to express that from the premises $P_1, \ldots, P_n$ we can conclude C. (Here, $P_1, \ldots, P_n$ and C are built from the object language involving $\longrightarrow$.) Backward proving of theorems is supported by tactics, which are single proof commands, and tacticals, which are proof strategies used to prove more complex proof commands.

There are the basic types *unit*, *bool*, *int* and *nat*, along with the polymorphic type $\alpha(set)$ of sets for any element of type α. Product types $\alpha \times \beta$ are provided with the additional selector functions fst and snd, whereas sum types $\alpha + \beta$ come with the injections Inl and Inr (*i.e.* $\alpha + \beta =$ Inl α | Inr β). The ternary sum $\alpha + \beta + \gamma$ has injections In1, In2 and In3. For nested sums, In1l e and In1r e are used to denote In1 (Inl e) and In1 (Inr e) respectively. The list type $\alpha(list)$ is defined by the datatype declaration $\alpha(list) = [] \mid \alpha\#(\alpha)list$, where [] denotes the empty list and # is the infix "cons" operator.

Logical constants are declared by a name and type separated by a "::" symbol. Primitive recursive function definitions are written in a standard way. Non-recursive definitions are written using the $\equiv$ symbol. Predicates are functions with boolean result and the functions themselves are written in a curried style. Hilbert's choice operator ε is used for descriptions: $\varepsilon x.\, P\, x$ denotes either some value x satisfying P, or an arbitrary value if no such x exists. In order to emulate partial functions the option type $\alpha(option)$ is often used. It is defined by the datatype declaration $\alpha(option) =$ None | Some α. The function the :: $(\alpha)option \to \alpha$ is defined whereby the (Some x) $= x$ and the None = arbitrary. Here arbitrary is an unknown value defined as $\varepsilon x.$ False.

Recall that a record r in Isabelle/HOL is a collection of n fields. We use different notation than that presented in Chapter 2; instead we denote a record between the brackets "(|" and "|)". (This is a convention recently introduced by users of Proof General, a generic interface for theorem provers [121].) A record is defined as follows: $r = (|c_1 :: \sigma_1, \ldots, c_n :: \sigma_n|)$. Each field has a name c_i, a specified type σ_i and a selector function of the same name. A record may be extensible: $r + (|c_{n+1} :: \sigma_{n+1}|)$ extends r with the field c_{n+1}.

We conclude this section with some typographic remarks. Following [111] we adopt sans serif for Isabelle constructors such as True or Expr. Type names like *bool* or *state*, and variables like v appear in italics. Java keywords such as `catch` or `Object` appear in typewriter font.

5.4 The Bali calculus

As outlined in [113] Bali includes the following features of Java: class and interface declarations with instance fields and methods; subinterfaces, subclasses and implementation relations with inheritance, overriding and hiding; method calls with static overloading and dynamic binding; some primitive types and objects (including arrays); and exception throwing and handling. It does not yet consider multi-threading. Recent developments in Bali have been Schirmer's formalisation of access concepts and definite assignment in Isabelle/HOL [129; 130]. These are discussed further in section 5.6. Bali's latest Isabelle theory files can be found at the project's website [8].

5.4.1 Syntax and semantics

In this section we look at Bali's abstract syntax and its operational semantics. The latter comes in the form of a state model along with an assortment of evaluation rules. We begin by first defining a Bali program.

Programs A Bali program is a record of lists of interface and class declarations.

$$prog = (\!|ifaces :: (idecl)list,\ classes :: (cdecl)list|\!)$$

Each interface and class declaration is a pair of a name and the defined entity. There exists a number of predefined names: for example the system exceptions *xname* listed below.

$$\begin{aligned} xname \quad &= \texttt{Throwable} \mid \texttt{Nullpointer} \mid \texttt{OutOfMemory} \\ &\mid \texttt{ClassCast} \mid \texttt{NegArrSize} \mid \texttt{IndOutBound} \mid \texttt{ArrStore} \end{aligned}$$

Note that Bali's exception names are abbreviated, *i.e.* `Nullpointer` abbreviates `NullPointer Exception` and `ArrStore` abbreviates `ArrayStoreException`, *etc.* The opaque HOL types *tnam*, *pname*, *vname* and *mname* respectively represent user-defined type, package, variable or field, and method names. The record *qtname* qualifies all type names for interfaces and classes with a package name.

tname	= `Object`	name of the top of the class hierarchy
	| SXcpt *xname*	name of a system exception
	| TName *tnam*	other class or interface name
qtname	= $(\!\|pid :: pname,\ tid :: tname\|\!)$	record of package and type names
lname	= This	special name for `this` pointer
	| EName *ename*	name for local variables
ename	= VNam *vname*	variable or field names
	| Res	name to model return value of methods

Before we define the components of a program we introduce the records *decl* and *member*. The first of these can be considered the "base" record of interface and class record declarations: it simply contains the access modifier of a class or interface member. The record *member* extends *decl* with the field *static* which flags whether or not the member is declared `static`. (Access and static modifiers are discussed further in Section 5.6.1.)

$$\begin{aligned} acc\_modi &= \text{Private} \mid \text{Package} \mid \text{Protected} \mid \text{Public} \\ decl &= (\!|access :: acc\_modi|\!) \\ stat\_modi &= bool \\ member &= decl + (\!|static :: stat\_modi|\!) \end{aligned}$$

An interface declaration *idecl* is a pair of a qualified type name *qtname* and an interface *iface*. The interface contains lists of superinterface names and method declarations. A class declaration *cdecl* is a pair of a *qtname* and a class *class*. The class specifies the names of a superclass and its implemented interfaces, lists of field and method declarations and a static initialiser *init*. The static initialiser is a block of code that is used to initialise the static variables of a class; in fact it only has access to static class variables. In Bali, all static initialisers of a

class are combined into a single block of type *stmt*.

$$\begin{array}{ll} idecl & = qtname \times iface \\ iface & = ibody + (\!|isuperIfs :: (qtname)list|\!) \\ ibody & = decl + (\!|imethods :: (sig \times mhead)list|\!) \\ cdecl & = qtname \times class \\ class & = cbody + (\!|super :: qtname,\ superIfs :: (qtname)list|\!) \\ cbody & = decl + (\!|cfields :: (fdecl)list,\ methods :: (mdecl)list,\ init :: stmt|\!) \end{array}$$

A field declaration *fdecl* gives the field name and type *ty* (see the section on types for a definition). A method declaration *mdecl* consists of: a signature, *i.e.* the method name and the list of parameter types excluding the result type; a method header *mhead* which consists of the list of parameter names and the result type; and (only if it appears within a class) the method body *mbody*. The method body is the lists of local variables and types along with the body statement itself (see the section on terms for a definition of the latter).

$$\begin{array}{ll} fdecl & = vname \times field \\ field & = member + (\!|type :: ty|\!) \\ mdecl & = sig \times methd \\ sig & = mname \times ty \\ methd & = mhead + (\!|mbody :: mbody|\!) \\ mhead & = member + (\!|pars :: (vname)list,\ resT :: ty|\!) \\ mbody & = (\!|lcls :: (vname \times ty)list,\ stmt :: stmt|\!) \end{array}$$

Types Bali types are formalised as values of datatype *ty* and appear as either primitive or reference types.

$$\begin{array}{lll} ty = \mathsf{PrimT}\ prim\_ty & prim\_ty = \texttt{void} & ref\_ty = \mathsf{NullT} \\ \quad | \mathsf{RefT}\ ref\_ty & \quad | \texttt{boolean} & \quad | \mathsf{IfaceT}\ qtname \\ & \quad | \texttt{int} & \quad | \mathsf{ClassT}\ qtname \\ & & \quad | \mathsf{ArrayT}\ ty \end{array}$$

There are three kinds of primitive types: boolean values, integers and the `void` type. The latter is used as a "dummy" type for methods that do not return a result. In Java, there are three kinds of reference type: interface, class and array types. For unification purposes, Bali adds the NullT type to this list. Furthermore the following abbreviations are often used.

$$\begin{array}{ll} \mathsf{NT} & \equiv \mathsf{RefT}\ \mathsf{NullT} \\ \mathsf{Iface}\ I & \equiv \mathsf{RefT}\ (\mathsf{IfaceT}\ I) \\ \mathsf{Class}\ C & \equiv \mathsf{RefT}\ (\mathsf{ClassT}\ C) \\ T[] & \equiv \mathsf{RefT}\ (\mathsf{ArrayT}\ T) \end{array}$$

Terms These are: statements, which appear in method bodies; expressions, which appear in statements; and values and variables, which appear in expressions as recursive datatypes. Statements are stripped to their bare essentials. There is no formalisation of the `switch` statement, nor of any jump statements such as `break` or `continue`. A formalism of `try_catch_finally` is constructed using the Try_Catch and Finally statements. A formalism of multiple `catch` clauses is constructed using cascaded If_Else statements and the InstOf expression (defined

below). It is also worth noting that nested blocks are not considered: a block in Bali is simply a statement that may contain other statements *via* sequential composition.

$$
\begin{array}{rl}
\mathit{stmt} & = \mathsf{Skip} \\
& \mid \mathsf{Expr}\ \mathit{expr} \\
& \mid \mathit{stmt};\ \mathit{stmt} \\
& \mid \mathsf{If}\ (\mathit{expr})\ \mathit{stmt}\ \mathsf{Else}\ \mathit{stmt} \\
& \mid \mathsf{While}\ (\mathit{expr})\ \mathit{stmt} \\
& \mid \mathsf{Throw}\ (\mathit{expr}) \\
& \mid \mathsf{Try}\ \mathit{stmt}\ \mathsf{Catch}\ (\mathit{qtname}\ \mathit{vname})\ \mathit{stmt} \\
& \mid \mathit{stmt}\ \mathsf{Finally}\ \mathit{stmt} \\
& \mid \mathsf{Init}\ \mathit{qtname}
\end{array}
$$

The statement Skip denotes the empty statement, whereas the constructor Expr is used to convert expressions to statements. In particular, assignments and method calls may be turned into statements and once these are evaluated they describe possible side-effects. It is sometimes the case that the first active use of a class C triggers its own initialisation. To model this behaviour the statement Init is used in the evaluation of a number of expressions.

Expressions are defined as follows:

expr	$= \mathsf{NewC}\ \mathit{qtname}$	class instance creation
	$\mid \mathsf{New}\ \mathit{ty}[\mathit{expr}]$	array creation
	$\mid \mathsf{Cast}\ \mathit{ty}\ \mathit{expr}$	type cast
	$\mid \mathit{expr}\ \mathsf{InstOf}\ \mathit{ref\_ty}$	type comparison operator
	$\mid \mathsf{Lit}\ \mathit{val}$	literal
	$\mid \mathsf{Super}$	special Super keyword
	$\mid \mathsf{Acc}\ \mathit{var}$	variable access
	$\mid \mathit{var} := \mathit{expr}$	variable assignment
	$\mid \mathit{expr}\,?\,\mathit{expr}\ :\ \mathit{expr}$	conditional
	$\mid \{\mathit{qtname}, \mathit{ref\_ty}, \mathit{inv\_mode}\}$	
	$\quad \mathit{expr} \cdot \mathit{mname}(\{(\mathit{ty})\mathit{list}\}(\mathit{expr})\mathit{list})$	method call
	$\mid \mathsf{Methd}\ \mathit{qtname}\ \mathit{sig}$	folded method
	$\mid \mathsf{Body}\ \mathit{qtname}\ \mathit{stmt}$	unfolded method body

The constructor NewA creates only one-dimensional arrays, however multi-dimensional array creation can be emulated with nested array creation. The constructor Super has the same value as This and has as its type the supertype of This. The constructor Acc, for variable access, is introduced to avoid syntactic ambiguities. The type annotations (in braces) declared in the method call provide auxiliary information for resolving method overloading and for the static binding of fields. A call is usually of the form $\{\mathit{accC}, \mathit{statT}, \mathit{inv\_mode}\}e \cdot \mathit{mn}(\{\mathit{pTs}\}\mathit{args})$. Here accC is the accessing class, *i.e.* the static class from which the method call is made, and statT is the static declaration class or interface of the method, namely the static type of e. The invocation mode $\mathit{inv\_mode}$ for the method call is one of the following: Static, SuperM, or IntVir. (Respectively, these formalise the invocation modes `static`, `super`, `interface` or `virtual`.) The expression e is a reference to an object, whereas mn is a field name. A list

of types of parameters is given by *pTs*, and *args* lists the actual parameters/arguments. Lastly the Methd and Body expressions are artificial program constructs. The Methd expression denotes the implementation of a method of a particular class. The unfolded version of a method implementation (denoted by the Body expression) is defined as the method's actual body. Both expressions are crucial to Bali's axiomatic semantics.

We next define Bali's values and variables.

val	= Unit	*var*	= LVar *lname*	local variable
	| Bool *bool*		| $\{qtname, qtname, bool\}$ *expr..vname*	class field
	| Intg *int*		| *expr*.[*expr*]	array component
	| Null			
	| Addr *loc*			

Values rely on the standard HOL types *bool* and *int*. The type *loc* of locations is not further specified. The value Unit serves as a dummy result of statements and `void` methods. With regards to variables, a typical class field has the form $\{accC, statDeclC, stat\}e..fn$ where *accC* is the accessing class, *statDeclC* is the static declaration class of the field, *stat* flags whether the field is either a static or an instance field, *e* is an object reference and *fn* is a field name. Again, a number of abbreviations are adopted.

`this`	≡ Acc (LVar This)	$!!v$	≡ Acc (LVar (EName v))
		$v :== e$	≡ Expr (LVar (EName v)) := e)

Lookup tables We conclude our discussion on Bali's abstract syntax by examining the representation of lookup tables in Bali. In order to look up declared entities, declaration lists are transformed into abstract tables which are modelled as partial functions. A table with key type α and entry type β is defined as follows:

$$(\alpha, \beta)table = \alpha \Rightarrow (\beta)option$$

The equality $t\ x =$ None has the meaning that there is no entry for key x in table t, whereas $t\ x =$ Some y means that x is associated with entry y. The empty table, the pointwise update of a table, and the function table_of which converts a declaration list into a table, are defined below.

empty	$:: (\alpha, \beta)table$
$\_[\_ \mapsto \_]$	$:: (\alpha, \beta)table \Rightarrow \alpha \Rightarrow \beta \Rightarrow (\alpha, \beta)table$
table_of	$:: (\alpha \times \beta)list \Rightarrow (\alpha, \beta)table$
empty	$\equiv \lambda k.$ None
$t[x \mapsto y]$	$\equiv \lambda k.$ if $k = x$ then Some y else $t\ k$
table_of[]	= empty
table_of$((k,x)\#t)$	= (table_of t)$[k \mapsto x]$

These definitions will appear in the sequel.

Heading towards a definition of the operational semantics of Bali, we now introduce the general notion of objects. We will then describe Bali's state model and give the evaluation rules for statements and expressions.

Objects An object is either a class instance or an array. The former is modelled as a pair of the instance's class name along with a table mapping pairs of a field name and the defining class to values. The latter is modelled as a pair of the array's component type along with a table mapping integers to values.

$$\begin{array}{ll} \mathit{obj} & = (\!|\mathit{tag} :: \mathit{obj\_tag},\ \mathit{values} :: (\mathit{vn}, \mathit{val})\mathit{table}|\!) \\ \mathit{obj\_tag} & = \mathsf{CInst}\ \mathit{qtname} \mid \mathsf{Arr}\ \mathit{ty}\ \mathit{int} \\ \mathit{vn} & = (\mathit{vname} \times \mathit{qtname}) + \mathit{int} \end{array}$$

A number of access functions are defined on objects: among them are upd_obj, obj_ty and obj_class. The function upd_obj updates a variable within an object, *i.e.* a field or array component. The function obj_ty returns the type of an object and obj_class returns the class to be used for a method call upon the given object. The three are defined as follows:

$$\begin{array}{ll} \mathsf{upd\_obj} & :: \mathit{vn} \Rightarrow \mathit{val} \Rightarrow \mathit{obj} \Rightarrow \mathit{obj} \\ \mathsf{obj\_ty} & :: \mathit{obj} \Rightarrow \mathit{ty} \\ \mathsf{obj\_class} & :: \mathit{obj} \Rightarrow \mathit{ty} \\ \\ \mathsf{upd\_obj}\ n\ v & \equiv \lambda(\mathit{oi}, \mathit{vs}).\ (\mathit{oi}, \mathit{vs}(n \mapsto v)) \\ \mathsf{obj\_ty}\ \mathit{obj} & \equiv \mathsf{case}\ \mathsf{fst}\ \mathit{obj}\ \mathsf{of}\ \mathsf{CInst}\ C \Rightarrow \mathsf{Class}\ C \mid \mathsf{Arr}\ T\ k \Rightarrow T[] \\ \mathsf{obj\_class}\ \mathit{obj} & \equiv \mathsf{case}\ \mathsf{fst}\ \mathit{obj}\ \mathsf{of}\ \mathsf{CInst}\ C \Rightarrow C \mid \mathsf{Arr}\ T\ k \Rightarrow \mathtt{Object} \end{array}$$

The store $(\mathit{qtname}, \mathit{obj})\mathit{table}$ is used to reference class objects containing static fields, whereas the store $(\mathit{loc}, \mathit{obj})\mathit{table}$ is used to reference ordinary objects *via* locations on the heap. Hence a generalised object reference is introduced, which is either a location or a class name, and is defined as follows:

$$\mathit{oref} = \mathit{loc} + \mathit{qtname}$$

Additionally, Heap is used to denote Inl and Stat, Inr.

States A state consists of an optional exception and two stores: one for (global) objects and one for local variables, including method and exception parameters and the This pointer. The stores are combined in an abstract datatype *st* representing the "pure" state, namely the contents of all the variables. The heap is also defined.

$$\begin{array}{ll} \mathit{state} & = \mathit{abopt} \times \mathit{st} \\ \mathit{abopt} & = (\mathit{abrupt})\mathit{option} \\ \mathit{st} & = \mathsf{st}\ \mathit{globs}\ \mathit{locals} \\ \mathit{globs} & = (\mathit{oref}, \mathit{obj})\mathit{table} \\ \mathit{locals} & = (\mathit{lname}, \mathit{val})\mathit{table} \\ \mathit{heap} & = (\mathit{loc}, \mathit{obj})\mathit{table} \end{array}$$

The various types of abrupt completions are classified below. The opaque HOL type *label*

represents the user-defined destination of a `break` or `continue` statement.

abrupt	= Xcpt *xcpt*	exception
	| Jump *jump*	break, continue and return
	| Error *error*	runtime errors
xcpt	= Loc *loc*	location of allocated exception object
	| Std *xname*	intermediate standard exception
jump	= Break *label*	`break`
	| Cont *label*	`continue`
	| Ret	`return` from a method
error	= AccessViolation	access to a member that is not permitted
	| CrossMethodJump	method exits with a break or continue

In order to manipulate the store a number of operations are introduced. These include: globs and locals for read access; and lupd and upd_gobj for update and set access. Their definitions rely on the functions st_case and chg_map.

globs	:: $st \Rightarrow globs$
locals	:: $st \Rightarrow locals$
lupd($\_ \mapsto \_$)	:: $lname \Rightarrow val \Rightarrow st \Rightarrow st$
upd_gobj	:: $oref \Rightarrow vn \Rightarrow val \Rightarrow st \Rightarrow st$
globs (st g l)	$\equiv g$
locals (st g l)	$\equiv l$
lupd($vn \mapsto v$) (st g l)	$\equiv$ st g ($l(vn \mapsto v)$)
upd_gobj r n v (st g l)	$\equiv$ st (chg_map (upd_obj n v) r g) l
chg_map f a m	$\equiv$ case m a of None $\Rightarrow$ m | Some $b \Rightarrow m(a \mapsto f\ b)$

There also exists a number of useful functions for accessing the heap.

heap	:: $st \Rightarrow heap$
lookup_obj	:: $st \Rightarrow val \Rightarrow obj$
heap s	$\equiv$ globs $s \circ$ Heap
lookup_obj s a	$\equiv$ the (heap s (the_Addr a))

The predicates inited and initd are used to check whether a given class has been initialised, *i.e.* whether or not its class object is available. Note that store $\sigma \equiv$ snd σ is used to represent the store of a state σ.

inited	:: $qtname \Rightarrow globs \Rightarrow bool$
initd	:: $qtname \Rightarrow state \Rightarrow bool$
inited C g	$\equiv g$ (Stat C) $\neq$ None
initd C σ	$\equiv$ inited C (globs (store σ))

Furthermore the two functions abupd and supd map an update of the exception or store part of a state to an update of the full state. They are defined as follows:

$$\begin{array}{ll} \mathsf{abupd} & :: (\mathit{abopt} \Rightarrow \mathit{abopt}) \Rightarrow \mathit{state} \Rightarrow \mathit{state} \\ \mathsf{supd} & :: (\mathit{st} \Rightarrow \mathit{st}) \Rightarrow \mathit{state} \Rightarrow \mathit{state} \\ \\ \mathsf{abupd}\ f\ (x,s) & \equiv (f\ x, s) \\ \mathsf{supd}\ f\ (x,s) & \equiv (x, f\ s) \end{array}$$

For a store s the abbreviation $\mathsf{Norm}\ s \equiv (\mathsf{None}, s)$ is used to represent normal (exception-free) states. Moreover the predicate $\mathsf{normal}\ \sigma \equiv (\mathsf{fst}\ \sigma = \mathsf{None})$ holds if state σ is normal. Often situations arise when an exception should be raised only if no exception is already present. If an exception is already present it should take precedence. Such behaviour is captured by the function abrupt_if.

$$\begin{array}{ll} \mathsf{abrupt\_if} & :: \mathit{bool} \Rightarrow \mathit{abopt} \Rightarrow \mathit{abopt} \Rightarrow \mathit{abopt} \\ \mathsf{abrupt\_if}\ c\ x'\ x & \equiv \mathsf{if}\ c \wedge (x = \mathsf{None})\ \mathsf{then}\ x'\ \mathsf{else}\ x \end{array}$$

The following abbreviations are commonly used.

$$\begin{array}{ll} \mathsf{raise\_if}\ c\ xn & \equiv \mathsf{abrupt\_if}\ c\ (\mathsf{Some}\ (\mathsf{Xcpt}\ (\mathsf{Std}\ xn))) \\ \mathsf{error\_if}\ c\ e & \equiv \mathsf{abrupt\_if}\ c\ (\mathsf{Some}\ (\mathsf{Error}\ e)) \\ \mathsf{np}\ v & \equiv \mathsf{raise\_if}\ (v = \mathsf{Null})\ \texttt{Nullpointer} \end{array}$$

Here $\mathsf{np}\ v$ propagates any present exception and otherwise throws the `Nullpointer` exception if the value v (which is assumed to be a reference) is the Null pointer.

Judgements In Bali, the judgements for the execution of statements and the evaluation of expressions, expression lists and variables, are all combined into one. A general evaluation judgement has the form below.

$$\mathit{prog} \vdash \mathit{state} - \mathit{term} \succ\!\!\rightarrow (\mathit{vals} \times \mathit{state})$$

The judgement $G \vdash \sigma - t \succ\!\!\rightarrow (w, \sigma')$ has the meaning that in the context of program G, evaluation of the term t (a statement, expression, value or variable) from the initial state σ terminates in a state σ' and yields the result w. Note that we deviate from von Oheimb by using G instead of Γ to denote a Bali program: this avoids confusion with the antecedent Γ in JavaCard DL sequents. We have the following syntactic variants.

$$\begin{array}{l} \mathit{prog} \vdash \mathit{state} - \mathit{stmt} \rightarrow \mathit{state} \\ \mathit{prog} \vdash \mathit{state} - \mathit{expr} \succ \mathit{val} \rightarrow \mathit{state} \\ \mathit{prog} \vdash \mathit{state} - \mathit{var} \Rightarrow \mathit{vvar} \rightarrow \mathit{state} \end{array}$$

A statement *stmt* is considered a special form of expression and its evaluation is assigned the dummy result value Unit. An expression *expr* evaluates to a value *val*. Variables *var* evaluate to a pair *vvar* consisting of a current value (for read access) and a state-transforming update

function which depends on the value to be assigned to the variable. The generalised result type for terms is defined as *vals*.

$$\begin{aligned} vvar &= val \times (val \Rightarrow state \Rightarrow state) \\ vals &= val + vvar + (val)list \end{aligned}$$

The syntactic variants are abbreviated below. Note that $\bullet \equiv \mathsf{In1}\ \mathsf{Unit}$.

$$\begin{aligned} G \vdash \sigma - c \rightarrow \sigma' &\equiv G \vdash \sigma - \mathsf{In1r}\ c \succ\!\!\rightarrow (\bullet, \sigma') \\ G \vdash \sigma - e \succ\!\!> v \rightarrow \sigma' &\equiv G \vdash \sigma - \mathsf{In1l}\ e \succ\!\!\rightarrow (\mathsf{In1}\ v, \sigma') \\ G \vdash \sigma - e \Rightarrow\!\!> vf \rightarrow \sigma' &\equiv G \vdash \sigma - \mathsf{In2}\ e \succ\!\!\rightarrow (\mathsf{In2}\ vf, \sigma') \end{aligned}$$

In his PhD thesis [111] von Oheimb explains why an evaluation, or "big-step" semantics for Bali has been deliberately chosen over a transition, or "small-step" semantics: first of all an evaluation semantics is easier to read because it is more abstract and less verbose. Secondly, it is easier to validate since the Java language specification is given in an evaluation-oriented operational style. Thirdly, within the more complex rules intermediate values need not be stored explicitly. And lastly, proofs are more easier to conduct since potentially problematic invariants on intermediate states within the execution of a single term are not required.

5.4.2 Aspects of the Bali calculus

In this section we discuss a number of evaluation rules that are of relevance to us, in particular the variable assignment rule. We refer the reader to [111; 113] for a more thorough listing of the rules. We begin by first presenting the evaluation rule for the sequential composition of statements c_1 and c_2.

$$\frac{G \vdash \mathsf{Norm}\, s_0 - c_1 \rightarrow \sigma_1 \qquad G \vdash \sigma_1 - c_2 \rightarrow \sigma_2}{G \vdash \mathsf{Norm}\, s_0 - c_1; c_2 \rightarrow \sigma_2} \tag{5.1}$$

Hence starting in a normal initial state $\mathsf{Norm}\ s_0$, a run of the sequential composition $c_1; c_2$ will terminate in a state σ_2 if a run of c_1 from $\mathsf{Norm}\ s_0$ terminates in σ_1 and a run of c_2 from σ_1 terminates in σ_2.

The often-used expression evaluation rule is as follows:

$$\frac{G \vdash \mathsf{Norm}\, s_0 - e \succ\!\!> v \rightarrow \sigma_1}{G \vdash \mathsf{Norm}\, s_0 - \mathsf{Expr}\, e \rightarrow \sigma_1} \tag{5.2}$$

The evaluation rule for the variable assignment expression $va := e$ is given below. It evaluates e and uses its value to possibly update the state. Recall that *va* is one of the following: a local variable $\mathsf{LVar}\ vn$, a class field $\{accC, statDeclC, stat\}e..fn$, or an array component $e_1.[e_2]$.

$$\frac{G \vdash \mathsf{Norm}\, s_0 - va \Rightarrow\!\!> (w, f) \rightarrow \sigma_1 \qquad G \vdash \sigma_1 - e \succ\!\!> v \rightarrow \sigma_2}{G \vdash \mathsf{Norm}\, s_0 - va := e \succ\!\!> v \rightarrow \mathsf{assign}\ f\ v\ \sigma_2} \tag{5.3}$$

The update takes place only if the following conditions are upheld: there is no exception already present; and the update function itself does not throw an exception. Such behaviour

is described by the function assign. If an exception is already present, assign will merely propagate it.

$$\begin{array}{ll} \mathsf{assign} & :: (val \Rightarrow state \Rightarrow state) \Rightarrow val \Rightarrow state \Rightarrow state \\ \mathsf{assign}\ f\ v & \equiv \lambda(x,s).\ \mathsf{let}\ (x',s') = \mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ f\ v\ (x,s)\ \mathsf{else}\ (x,s) \\ & \qquad \mathsf{in}\ (x', \mathsf{if}\ x' = \mathsf{None}\ \mathsf{then}\ s'\ \mathsf{else}\ s) \end{array}$$

Regarding the evaluation of variables themselves we have seen previously that they evaluate to *vvar* which consists of the current value of the variable and an update function. (Recall that a variable is defined as one of the following: a local variable LVar *vn*, a class field $\{accC, statDeclC, stat\}e..fn$, or an array component $e_1.[e_2]$.) Due to the complicated nature of this function, three different rules are defined, one for each variable. The simplest of these is the local variable evaluation rule.

$$\frac{}{G \vdash \mathsf{Norm}\ s - \mathsf{LVar}\ vn \Rightarrow\!\!\succ \mathsf{lvar}\ vn\ s \rightarrow \mathsf{Norm}\ s} \tag{5.4}$$

Starting in an initial normal state Norm *s*, evaluation of the local variable LVar *vn* will yield the result lvar *vn s* and terminate in the state Norm *s*, *i.e.* the evaluation will cause no side-effects. The function lvar is defined as follows:

$$\begin{array}{ll} \mathsf{lvar} & :: lname \Rightarrow st \Rightarrow vvar \\ \mathsf{lvar}\ vn\ s & \equiv (\mathsf{the}\ (\mathsf{locals}\ s\ vn),\ \lambda v.\ \mathsf{supd}\ (\mathsf{lupd}\ (vn \mapsto v))) \end{array}$$

The evaluation rule for field variables is somewhat more complicated.

$$\frac{\begin{array}{c} G \vdash \mathsf{Norm}\ s_0 - \mathsf{Init}\ statDeclC \rightarrow \sigma_1 \\ G \vdash \sigma_1 - e\!\!\succ\ a \rightarrow \sigma_2 \\ (v,\sigma_3) = \mathsf{fvar}\ statDeclC\ stat\ fn\ a\ \sigma_2 \\ \sigma_4 = \mathsf{check\_field\_access}\ G\ accC\ statDeclC\ fn\ stat\ a\ \sigma_3 \end{array}}{G \vdash \mathsf{Norm}\ s_0 - \{accC, statDeclC, stat\}e..fn \Rightarrow\!\!\succ v \rightarrow \sigma_4} \tag{5.5}$$

Within the evaluation, the first active use of the class *statDeclClass* possibly triggers its own initialisation. This behaviour is captured by the Init statement being evaluated appropriately. Its own evaluation rule is as follows:

$$\frac{\begin{array}{l} \mathsf{the}\ (\mathsf{class}\ G\ C) = c \\ \mathsf{if\ inited}\ C\ (\mathsf{globs}\ s_0)\ \mathsf{then}\ \sigma_3 = \mathsf{Norm}\ s0 \\ \mathsf{else}\ (G \vdash \mathsf{Norm}\ (\mathsf{init\_class\_obj}\ G\ C\ s_0) \\ \qquad - (\mathsf{if}\ C = \mathsf{Object}\ \mathsf{then\ Skip\ else\ Init}\ (\mathsf{super}\ c)) \rightarrow \sigma_1 \\ \quad \wedge\ G \vdash \mathsf{set\_lvars\ empty}\ \sigma_1 - \mathsf{init}\ c \rightarrow \sigma_2 \wedge \sigma_3 = \mathsf{restore\_lvars}\ \sigma_1\ \sigma_2) \end{array}}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Init}\ C \rightarrow \sigma_3} \tag{5.6}$$

If class *C* has already been initialised, *i.e.* inited returns True then nothing happens. If *C* has not been initialised, then – without going into too much detail – a new class object is allocated *via* init_class_obj (see [111] for a definition). If the class is not `Object`, its superclass is initialised. The static initialiser init of the current class is then executed and (since it may only have access

to static class variables) the current local variables are first hidden and then restored.

Hence returning to our description of the field variable evaluation rule: starting in an initial state Norm s_0, evaluation of the field variable $\{accC,\ statDeclC,\ stat\}e..fn$ will yield the value v and terminate in the final state σ_4. It is assumed that: (1) the static declaration class *statDeclC* of the field has been initialised either sometime in the past or immediately before e is evaluated; (2) the expression e evaluates to address a; (3) the value of the field is found using the function fvar and either the state is updated or a null pointer exception is thrown if a is Null; and (4) a test is conducted (using the function check_field_access) to see whether the current field is dynamically accessible – an error is thrown otherwise. (See section 5.6.1 for further discussion on dynamic accessibility and the check_field_access function.) The function fvar is defined below. Note that id is the Isabelle/HOL identity function.

$$
\begin{array}{ll}
\mathsf{fvar} & :: \mathit{qtname} \Rightarrow \mathit{bool} \Rightarrow \mathit{vname} \Rightarrow \mathit{val} \Rightarrow \mathit{state} \Rightarrow \mathit{vvar} \times \mathit{state} \\
\mathsf{fvar}\ C\ \mathit{stat}\ \mathit{fn}\ a\ \sigma & \equiv \mathsf{let}\ (\mathit{oref}, \mathit{xf}) = \mathsf{if}\ \mathit{stat}\ \mathsf{then}\ (\mathsf{Stat}\ C, \mathsf{id})\ \mathsf{else}\ (\mathsf{Heap}\ (\mathsf{the\_Addr}\ a), \mathsf{np}\ a); \\
 & \qquad n = \mathsf{Inl}\ (\mathit{fn}, C);\ f = (\lambda v.\ \mathsf{supd}\ (\mathsf{upd\_gobj}\ \mathit{oref}\ n\ v)) \\
 & \quad \mathsf{in}\ ((\mathsf{the}\ (\mathit{values}\ (\mathsf{the}\ (\mathsf{globs}\ (\mathsf{store}\ \sigma)\ \mathit{oref}))\ n), f), \mathsf{abupd}\ \mathit{xf}\ \sigma)
\end{array}
$$

Depending on the flag *stat*, the field is either a static field of C or an instance field of an object. For static fields, the object reference is C itself. For instance fields, the reference needs to be looked up in the heap at address a. Recall that *values* is a field name (and hence selector function) of the record *obj*. It returns a table mapping pairs of a field name and the defining class to values.

The evaluation rule for array variables is as follows:

$$
\frac{G \vdash \mathsf{Norm}\ s_0 - e_1 \twoheadrightarrow a \rightarrow \sigma_1 \qquad G \vdash \sigma_1 - e_2 \twoheadrightarrow i \rightarrow \sigma_2 \qquad (v, \sigma_2') = \mathsf{avar}\ G\ i\ a\ \sigma_2}{G \vdash \mathsf{Norm}\ s_0 - e_1.[e_2] \Rightarrow v \rightarrow \sigma_2'} \tag{5.7}
$$

Here

$$
\begin{array}{ll}
\mathsf{avar} & :: \mathit{prog} \Rightarrow \mathit{val} \Rightarrow \mathit{val} \Rightarrow \mathit{state} \Rightarrow \mathit{vvar} \times \mathit{state} \\
\mathsf{avar}\ G\ i'\ a\ \sigma & \equiv \mathsf{let}\ \mathit{oref} = \mathsf{Heap}\ (\mathsf{the\_Addr}\ a);\ i = \mathsf{the\_Intg}\ i'; \\
 & \qquad n = \mathsf{Inr}\ i;\ (T, k, cs) = \mathsf{the\_Arr}\ (\mathsf{globs}\ (\mathsf{store}\ \sigma)\ \mathit{oref}); \\
 & \qquad f = (\lambda v\ (x, s). \\
 & \qquad\qquad (\mathsf{raise\_if}\ (\neg G, s \vdash v\ \mathsf{fits}\ T)\ \mathtt{ArrStore}\ x, \mathsf{upd\_gobj}\ \mathit{oref}\ n\ v\ s)) \\
 & \quad \mathsf{in}\ ((\mathsf{the}\ (cs\ n), f), \mathsf{abupd}\ (\mathsf{raise\_if}\ (\neg i\ \mathsf{in\_bounds}\ k)\ \mathtt{IndOutBound} \circ \mathsf{np}\ a)\ \sigma)
\end{array}
$$

As well as updating the state, the function avar checks for Null pointer de-referencing and for possible index bound violations (using in_bounds). Moreover it performs a dynamic type check on the value to be stored *via* the predicate fits. This will later play an important role in the proof of our translation of the KeY array assignment rule. Note that $G \vdash S \preceq T$ has the

meaning that S is a syntactic subtype of T in the context of program G.

$$\begin{array}{ll} \mathsf{in\_bounds} & :: \mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{bool} \\ i\ \mathsf{in\_bounds}\ k & \equiv 0 \leq i \wedge i < k \\ \\ \_,\_\vdash \_\ \mathsf{fits}\ \_ & :: \mathit{prog} \Rightarrow \mathit{st} \Rightarrow \mathit{valval} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool} \\ G,s \vdash a\ \mathsf{fits}\ T & \equiv (\exists \mathit{rt}.\ T = \mathsf{RefT}\ \mathit{rt}) \longrightarrow a = \mathsf{Null} \vee G \vdash \mathsf{obj\_ty}\ (\mathsf{lookup\_obj}\ s\ a) \preceq T \end{array}$$

Given this introduction to both the KeY and Bali calculi, we now proceed with describing the translation of the KeY assignment rules into Bali and outline their corresponding proofs in Isabelle/HOL. We address each assignment rule individually.

5.5 Local variable assignment

As provided by internal documentation, local variable assignment in the KeY tool is implemented as follows:

$$\langle \pi\ \mathrm{v} = \mathit{val};\ \omega \rangle \phi \ \rightsquigarrow\ \{\mathrm{v} := \mathit{val}\} \langle \pi\ \omega \rangle \phi$$

Here v is a local variable and *val* is a side-effect free expression. Hence the formula on the left expresses that $\pi\ \mathrm{v} = \mathit{val}$ sequentially composed with the "rest" of the program ω terminates normally in a state in which the JavaCard DL formula ϕ holds. This is transformed into a formula which is valid if, from a state which has been updated by $\{\mathrm{v} := \mathit{val}\}$, a run of the program $\pi\ \omega$ terminates in a state in which ϕ holds.

We translate this into the following Bali rule.

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathit{val} \succ\!\!\succ a \rightarrow \mathsf{Norm}\ s_0 \qquad G \vdash (\mathsf{None}, \mathsf{lupd}(\mathrm{v} \mapsto a)\ s_0) - \omega \rightarrow \sigma_1}{G \vdash \mathsf{Norm}\ s_0 - \mathrm{v} :== \mathit{val};\ \omega \rightarrow \sigma_1}$$

Since *val* is side-effect free, its evaluation (to value a) will not change the state in any way.

This rule is proven in Isabelle/HOL by first applying the rule for the sequential composition of statements (5.1) to the conclusion. After simplifying we find that we then need to prove $G \vdash \mathsf{Norm}\ s_0 - \mathrm{v} :== \mathit{val} \rightarrow \mathsf{Norm}\ (\mathsf{lupd}(\mathrm{v} \mapsto a)\ s_0)$ using our assumptions. To this new goal we apply the expression evaluation rule (5.2) followed by the evaluation rule for variable assignment (5.3), instantiating w as $\mathsf{the}\ (\mathsf{locals}\ s_0\ \mathrm{v})$ and f as $\lambda x.\ (\mathsf{supd}\ (\mathsf{lupd}(\mathrm{v} \mapsto x)))$. After further simplification we are left with the goal $G \vdash \mathsf{Norm}\ s_0 - \mathsf{LVar}\ \mathrm{v} =\!\!\succ (\mathsf{the}\ (\mathsf{locals}\ s_0\ \mathrm{v}), \lambda x.\ \mathsf{supd}\ (\mathsf{lupd}(\mathrm{v} \mapsto x))) \rightarrow \mathsf{Norm}\ s_0$. This is proven by an application of the local variable evaluation rule (5.4) and the definition of the function lvar. Our first proof is reasonably simple, but this situation will change with the field and array assignment rules.

5.6 Field variable assignment

We are looking to prove correct KeY's implementation of field assignment. We provide a translation of the implementation into Bali and prove it correct using Isabelle/HOL. However, before we address this, we need to discuss both access concepts and definite assignment analysis in Bali [129; 130]. Schirmer's work in these areas has recently been incorporated into von

Oheimb's original Bali formalism [111]. A number of Schirmer's results feature in our Isabelle proof of the translated JavaCard DL field assignment rule.

5.6.1 Bali access concepts

To recap: a Java program is simply a collection of interfaces and classes arranged in packages. Java access modifiers provide a means of controlling access to members (*i.e.* the fields or methods) of a class or interface and also to the class or interface itself [6]. An ordering of these modifiers, from the most restrictive to the most liberal is defined as follows:

$$\texttt{private} < \textit{package} < \texttt{protected} < \texttt{public}$$

Members declared `private` are accessible only in the class itself. Members declared with no access modifier are accessible in classes of the same package, as well as in the class itself. Members declared `protected` are accessible in subclasses of the class, in classes in the same package, and in the class itself. Members declared `public` are accessible anywhere the class is accessible.

In the Bali formalism a Java program is a mapping from qualified type names to the structures describing the corresponding classes and interfaces themselves. Consequently, there is an accessibility concept on the level of types as well as on the level of members. (The following Bali access definitions are taken from [129].) First, predicate $G \vdash T$ accessible_in P states that in the context of program G the type T is accessible in package P. Second, $G \vdash m$ member_of C states that m is a member of class C in the context of program G. A member, in this case, is a pair of type *qtname* $\times$ *memberdecl* consisting of the declaration class of the member and the member declaration itself. The functions declclass and mbr are used to select the components. The member declaration is such that *memberdecl* $=$ *fdecl* (*vname*,*field*) | *mdecl* (*sig*,*methd*). (The field declaration *fdecl* and method declaration *mdecl* are defined on page 88.) Hence a member declaration includes such information as: the identifier of the member; its access modifier; the type of the member; or whether or not it is a static or instance member. (Members may be declared `static`, meaning that they belong to a class and are not associated with a particular instance of that class.) We have the following rule, whereby the two functions mbr_declared_in_ and _member_of_ (similarly, _accessible_in_) take two arguments each.

$$\frac{G \vdash \mathsf{mbr}\, m\, \mathsf{declared\_in}\, (\mathsf{declclass}\, m)}{G \vdash m\, \mathsf{member\_of}\, (\mathsf{declclass}\, m)}$$

Here, the predicate $G \vdash \mathsf{mbr}\, m\, \mathsf{declared\_in}\, (\mathsf{declclass}\, m)$ demands that the declaration $\mathsf{mbr}\, m$ is present in the body of the declaration class of m.

The function Field enables us to talk of fields as members. Note that $\mathsf{fld} \equiv \mathsf{snd}$.

$$\begin{array}{ll} \mathsf{Field} & :: \textit{vname} \Rightarrow (\textit{qtname} \times \textit{field}) \Rightarrow (\textit{qtname} \times \textit{memberdecl}) \\ \mathsf{Field}\, n\, f & \equiv (\mathsf{declclass}\, f, \textit{fdecl}\, (n, \mathsf{fld}\, f)) \end{array}$$

The function accmodi returns the access modifier of a member m, whereas the predicate $G \vdash m$ in C permits_acc_from $accC$ expresses the access restrictions associated with the modifiers. For example, a member m declared `private` in class C permits access from an

accessing class *accC* when declclass m = *accC*. If m in C is declared `public` then $G \vdash m$ in C permits_acc_from *accC* is always True. The static accessibility of a member is defined using the predicate $G \vdash m$ of C accessible_from *accC*. This has the meaning that in the context of program G, the member m of class C is statically accessible from class *accC*. (Note that both functions _of_accessible_from_ and _in_permits_acc_from_ take three arguments.) Statically valid member access is determined by the rule

$$\frac{G \vdash m \text{ member\_of } C \quad G \vdash m \text{ in } C \text{ permits\_acc\_from } accC \quad G \vdash \text{Class } C \text{ accessible\_in } pid\ accC}{G \vdash m \text{ of } C \text{ accessible\_from } accC}$$

Recall that the record name *pid* returns the package name of a given class or interface.

As explained in [129] it is often the case in an object-oriented setting that if we statically expect a reference to an object of class A, we are able to receive an object of class B at runtime, where B is a subclass of A. In Java however, it is possible that A is declared `public` but B is not. Furthermore, even though B is not statically accessible, we still may be able to receive an object of B outside of its packages. Since accessibility of the class is a precondition for static accessibility of a member, we cannot always expect that at runtime only the statically accessible members are the members valid to access. Subsequently "dynamic" runtime accessibility is introduced to the Bali formalism. It is captured *via* the predicate $G \vdash m$ in C dyn_accessible_from *accC*, where the function _in_dyn_accessible_from_ takes three arguments. This states that in the context of program G, a member m of C is dynamically accessible from *accC*. Dynamically valid member access is determined by the rule

$$\frac{G \vdash m \text{ member\_in } C \quad G \vdash m \text{ in } C \text{ permits\_acc\_from } accC}{G \vdash m \text{ in } C \text{ dyn\_accessible\_from } C}$$

Note that a member is said to be "in" a class if it is a member "of" either the class itself or a member of a superclass, *i.e.* $G \vdash m$ member_in $C \equiv \exists C'.\ G \vdash C \preceq_c C' \wedge G \vdash m$ member_of C', where $\preceq_c$ is the subclass relation.

Schirmer's main theorem in [129] ensures that for a well-formed program (where only statically accessible members are accessed at compile-time) only dynamically accessible members are accessed at runtime. Dynamic accessibility is integrated into the operational semantics of Bali as special tests. These tests will cause the Bali defined error AccessViolation whenever dynamic accessibility is violated at runtime. As an example, we describe the function check_field_access used in the evaluation rule for field variables. It is somewhat complicated, but it is worth discussing here since we will later encounter it when proving the correctness of

KeY's implementation of field assignment. The function is defined as follows:

$$
\begin{array}{l}
\mathsf{check\_field\_access}\ G\ accC\ statDeclC\ fn\ stat\ a\ \sigma \equiv \\
\quad \mathsf{let}\ oref = \mathsf{if}\ stat\ \mathsf{then}\ \mathsf{Stat}\ statDeclC\ \mathsf{else}\ \mathsf{Heap}\ (\mathsf{the\_Addr}\ a); \\
\qquad dynC = \mathsf{case}\ oref\ \mathsf{of}\ \mathsf{Heap}\ a \Rightarrow \mathsf{obj\_class}\ (\mathsf{the}\ (\mathsf{globs}\ (\mathsf{store}\ \sigma)\ oref)) \\
\qquad\qquad\qquad\qquad \mid \mathsf{Stat}\ C \Rightarrow C; \\
\qquad f = (\mathsf{the}\ (\mathsf{table\_of}\ (\mathsf{fields}\ G\ dynC)(fn, statDeclC))) \\
\quad \mathsf{in}\ \mathsf{abupd}\ (\mathsf{error\_if}\ (\neg G \vdash \mathsf{Field}\ fn\ (statDeclC, f)\ \mathsf{in}\ dynC\ \mathsf{dyn\_accessible\_from}\ accC) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{AccessViolation})\ \sigma
\end{array}
$$

Depending on the flag *stat*, the field is either a static field of class *statDeclC*, or an instance field of an object. For static fields, the object reference is just the class name *statDeclC* itself. For instance fields, the reference needs to be looked up in the heap at address a. The dynamic class *dynC* of the reference is either stored in the heap for instance fields, or in the class *statDeclC* itself for static fields. The field f with (the extended) field name $(fn, statDeclC)$ is looked up in the field map of the dynamic class *dynC*. (The list fields $G\ C$ lists all the fields of a class C – including all fields of superclasses – in the context of a program G. Its definition relies on a general recursion operator for class hierarchies; we will not include it here.) Finally, the function check_field_access throws the error AccessViolation if dynamic accessibility is violated. The function error_if performs the test and abupd updates the state σ according to the outcome of the test.

In order to prove that no errors can occur during field variable evaluation, we need to prove that $G \vdash \mathsf{Field}\ fn\ (statDeclC, f)$ in *dynC* dyn_accessible_from *accC* holds at runtime. The rule given below shows us the conditions under which this predicate holds true. The reason that we introduce it here is because in order to prove correct the Bali translation of the KeY field assignment rule, we will need to prove that dynamic accessibility is not violated at runtime. Because KeY does not take into account such accessibility concepts, we will need to add extra assumptions to the translations that are not immediately obvious.

Schirmer's "dynamic field access OK" rule is as follows:

$$
\frac{\begin{array}{c}
\mathsf{wf\_prog}\ G \\
(G, L) \vdash e :: -\mathsf{Class}\ statC \\
\sigma ::\preceq (G, L) \\
G, (\mathsf{store}\ \sigma) \vdash a ::\preceq \mathsf{Class}\ statC \\
\neg stat \rightarrow a \neq \mathsf{Null} \\
\mathsf{normal}\ \sigma \\
\mathsf{accfield}\ G\ accC\ statC\ fn = \mathsf{Some}\ f \\
\mathsf{if}\ stat\ \mathsf{then}\ dynC = \mathsf{declclass}\ f\ \mathsf{else}\ dynC = \mathsf{obj\_class}\ (\mathsf{lookup\_obj}\ (\mathsf{store}\ s)\ a) \\
\mathsf{if}\ stat\ \mathsf{then}\ (\mathsf{is\_static}\ f)\ \mathsf{else}\ (\neg \mathsf{is\_static}\ f)
\end{array}}{\begin{array}{c}
\mathsf{table\_of}\ (\mathsf{fields}\ G\ dynC)(fn, \mathsf{declclass}\ f) = \mathsf{Some}\ (\mathsf{fld}\ f) \\
\wedge\ G \vdash \mathsf{Field}\ fn\ f\ \mathsf{in}\ dynC\ \mathsf{dyn\_accessible\_from}\ accC
\end{array}}
$$

First of all the program G must be well-formed. In Bali, the notion of well-formedness is defined for classes, interfaces, members of classes and interfaces, and programs. Well-formedness reflects the static global sanity checks made by the compiler for all declarations

at runtime. We refer the reader to [111] for definitions and a more detailed discussion. Secondly, the predicate $(G,L) \vdash e :: -\mathsf{Class}\ statC$ has the meaning that expression e is of static type $\mathsf{Class}\ statC$ and is well-typed in the environment (G,L). An environment is a pair of the program and the local environment. The local environment gives the types of the current local variables including the This pointer (in non-static methods) and method parameters. Next, $\sigma ::\preceq (G,L)$ expresses that state σ "conforms" to environment (G,L). This implies that all values within the state are compatible with their static types. Additionally $G, (\mathsf{store}\ \sigma) \vdash a ::\preceq \mathsf{Class}\ statC$ says that relative to the program G and store of σ, the address a conforms to the type $\mathsf{Class}\ statC$. This has the meaning that the dynamic type of a is a syntactic subtype of $\mathsf{Class}\ statC$. Furthermore $\neg stat \rightarrow a \neq \mathsf{Null}$ tells us that if the field is an instance field of an object, then the address a cannot be Null. We make the assumption that state σ is normal. Similar to the function fields, accfield $G\ C\ C'$ tables the fields of a class C which are accessible from C' in the context of program G. Hence $\mathsf{accfield}\ G\ accC\ statC\ fn = \mathsf{Some}\ f$ has the meaning that the lookup of the field name fn yields the field f. Next, depending on the flag *stat*, the dynamic class *dynC* is either the declaration class of f or else it is stored in the heap. Lastly, the field f is static (represented by $\mathsf{is\_static}\ f$) depending on the flag *stat*.

We now briefly discuss definite assignment in Bali before moving on to the translation of the KeY field assignment rule.

5.6.2 Bali definite assignment

According to the Java language specification [61] each local variable must have a definitely assigned value when any access of its value occurs. The Java compiler carries out a data flow analysis, making sure that for every access of a local variable, the variable is definitely assigned before the access; otherwise a compile-time error is thrown. In [130] the definite assignment analysis of the Java compiler is formalised in Isabelle/HOL (as part of the Bali calculus) and proven correct.

A relation $B \gg t \gg A$ is defined whereby B is the set of definitely assigned local variables before evaluation of the Bali term t, and A is the set of definitely assigned variables after its evaluation. Hence if the term t is evaluated from a state σ, then the predicate $(G,L) \vdash \mathsf{dom}\ (\mathsf{locals}\ (\mathsf{store}\ \sigma)) \gg t \gg A$ has the meaning that in the context of environment (G,L), term t has passed the definite assignment analysis. The already assigned variables in the current state σ are the input variables for the analysis; these are found by taking the domain – using the Isabelle/HOL domain mapping dom – of the local variable map locals of the store in σ.

5.6.3 KeY implementation and translation

KeY's implementation of the field assignment rule is given as follows:

$$\langle \pi\ o.\mathsf{a} := val;\ \omega \rangle \phi \rightsquigarrow \begin{cases} o = null \rightarrow \langle NPE \rangle \langle \pi\ \omega \rangle \phi \\ o \neq null \rightarrow \{o.\mathsf{a} := val\} \langle \pi\ \omega \rangle \phi \end{cases}$$

Here o and val are side-effect free expressions. The formula on the left expresses that the assignment $o.\mathsf{a} := val$ sequentially composed with ω terminates normally in a state in which the JavaCard DL formula ϕ holds. This is transformed into one of two formulae. The first is valid

if a `NullPointerException` is thrown from a state satisfying $o = null$, *i.e* the reference that accesses o is `null`. Unless the exception is caught further on in ω, the exception is continuously propagated. The second formula is valid if, from a state satisfying $o \neq null$ in which o.a has been updated by *val*, a run of the program $\pi\ \omega$ terminates in a state in which ϕ holds.

We translate this directly into the following Bali rule.

$$\frac{\begin{array}{l}\mathsf{initd}\ \mathit{statDeclC}\ (\mathsf{Norm}\ s_0) \\ G \vdash \mathsf{Norm}\ s_0 - o \twoheadrightarrow b \rightarrow \mathsf{Norm}\ s_0 \\ G \vdash (\mathsf{np}\ b\ \mathsf{None}, s_0) - \mathit{val} \twoheadrightarrow c \rightarrow \mathsf{Norm}\ s_0 \\ G \vdash (\mathsf{if}\ (b = \mathsf{Null}) \\ \quad \mathsf{then}\ (\mathsf{Xcpt}\ (\mathsf{Std}\ \texttt{Nullpointer}), s_0) \\ \quad \mathsf{else}\ (\mathsf{None}, (\mathsf{upd\_gobj}\ (\mathsf{if}\ \mathit{stat}\ \mathsf{then}\ (\mathsf{Stat}\ \mathit{statDeclC}) \\ \qquad\qquad \mathsf{else}\ (\mathsf{Heap}\ (\mathsf{the\_Addr}\ b))) \\ \qquad\qquad\qquad (\mathsf{Inl}\ (\mathsf{a}, \mathit{statDeclC}))\ c\ s_0))) - \omega \rightarrow \sigma_1\end{array}}{G \vdash \mathsf{Norm}\ s_0 - (\mathsf{Expr}\ (\{\mathit{accC}, \mathit{statDeclC}, \mathit{stat}\} o..\mathsf{a} := \mathit{val})); \omega \rightarrow \sigma_1}$$

The predicate initd checks whether a given class has already been initialised, or if initialisation is at least in progress. Hence the first assumption initd *statDeclC* (Norm s_0) tells us that at state Norm s_0 the class *statDeclC* has already been initialised and its class objects are available. Since KeY assumes that all classes are properly initialised, we are justified in adding this assumption to our Bali translation of the KeY implementation.

Next we assume o evaluates to the value b from the normal state Norm s_0. Because o is a side-effect free expression, there is no state change caused by the evaluation. Next, we assume the expression *val* evaluates to value c from state (None, s_0) as long as $b \neq$ Null. If b is Null then a `Nullpointer` exception is thrown. This situation is reproduced in the final assumption: if b is Null then the `Nullpointer` is propagated. If $b \neq$ Null then the state (store) is updated. The update takes into account whether the field is a static field of *statDeclC*, or an instance field of an object.

The conclusion $G \vdash$ Norm $s_0 -$ (Expr ($\{accC, statDeclC, stat\} o..\mathsf{a} := val$)); $\omega) \rightarrow \sigma_1$ has the meaning that evaluating the composed statement (Expr ($\{accC, statDeclC, stat\} o..\mathsf{a} := val$)); ω in the context of program G from the initial state Norm s_0, terminates in the state σ_1 (and yields the dummy result $\bullet$). Recall from the definition of field variables that *accC* represents the accessing class of field f, *statDeclC* is the declaration class of f and *stat* flags whether f is static or not.

It is important to note that the rule we have described above is incomplete. In order to prove it correct using Isabelle/HOL, we need to make some additional assumptions. The five assumptions listed below enable us to apply the dynamic field access OK rule; they ensure that dynamic accessibility is not violated during field variable evaluation, and that all local variables have a definitely assigned value when any access to its values occurs.

1. wf_prog G
2. $(G, L) \vdash o :: -\mathsf{Class}\ \mathit{statC}$
3. Norm $s_0 ::\preceq (G, L)$
4. $(G, L) \vdash \mathsf{dom}\ (\mathsf{locals}\ s_0) \gg \mathsf{In1l}\ o \gg A$

5. let $f = (statDeclC,$ the (table_of (fields G
(if $stat$ then $statDeclC$
else (obj_class (the (globs (store (Norm s_0))(Heap (the_Addr b))))))
)(a, $statDeclC$)))
in ((accfield G $accC$ $statC$ a = Some f)
$\wedge$ (if $stat$ then is_static f else $\neg$is_static f))

First of all the program G needs to be well-formed. Second, we require that o is of static type Class $statC$ and that is it well-typed in the environment (G,L). Third, the state Norm s_0 must conform to the environment (G,L). Fourth, o must pass the definite assignment analysis. Fifth, the assumption is comprised of two parts: the first part, accfield G $accC$ $statC$ a = Some f tells us that the lookup of the field name a – from the table of fields in $accC$ which are accessible from $statC$ – yields the field f, which is either a static field of class $statDeclC$ or an instance field of an object. The second part of the assumption tells us whether f is static depending on the flag $stat$.

Bali's *raison d'être* is to prove the type-safety and formal correctness of the Java programming language. It needs to formalise such concepts as dynamic accessibility and definite assignment in order to justify or invalidate claims made by Java's designers. KeY on the other hand assumes that the Java program to be verified is well-typed and that Java is type-safe, *i.e.* its evaluation will not result in an AccessViolation. We believe that adding these additional assumptions to our translated rule does not compromise the rule, since these program aspects (dynamic accessibility is not violated at runtime, local variables are definitely assigned before an access) are intrinsically assumed as part of the specification.

To prove the rule correct we consider two cases. The first case $b =$ Null is trivially proven. In the Bali formalism, when an exception is thrown, any subsequent computation is skipped and the exception is propagated until either it is caught or the program terminates. Hence exceptions are assumed to be present only in the final state of a judgement, never in the initial state. Therefore our assumption $G \vdash (\text{StdXcpt Nullpointer}, s_0) - \omega \rightarrow \sigma_1$ is always False.

For the case $b \neq$ Null we consider the two cases where either the field is static, or it is an instance field of an object. If $stat$, we apply the evaluation rule for sequential composition to the conclusion (5.1). We then need to prove the following:

$$G \vdash \text{Norm } s_0 - \text{Expr } (\{C, \text{True}\}o..\text{a} := val) \rightarrow \text{Norm } (\text{upd\_gobj } (\text{Stat } statDeclC) \\ (\text{Inl } (\text{a}, statDeclC))\, c\, s_0)$$

We do this by first applying the expression evaluation rule (5.2) followed by the evaluation rule for the variable assignment expression (5.3), instantiating w and f by W and F which we define as follows:

$$\begin{aligned} W &\equiv \text{the } (\text{values } (\text{the } (\text{globs } s_0\ (\text{Stat } statDeclC)))(\text{Inl } (\text{a}, statDeclC))) \\ F &\equiv \lambda x.\, \text{supd } (\text{upd\_gobj}(\text{Stat } statDeclC)(\text{Inl } (\text{a}, statDeclC))\, x) \end{aligned}$$

After simplification we are left with the following goal.

$$G \vdash \text{Norm } s_0 - \{C, \text{True}\}o..\text{a} \Rightarrow (W,F) \rightarrow \text{Norm } s_0$$

This can be proven by an application of the field variable evaluation rule (5.5) and the definition of the function fvar. However, before we can apply this rule, we need to prove the following two subgoals.

$$G \vdash \mathsf{Norm}\, s_0 - \mathsf{Init}\, statDeclC \rightarrow \mathsf{Norm}\, s_0$$
$$\mathsf{Norm}\, s_0 = \mathsf{check\_field\_access}\, G\, accC\, statDeclC\, \mathsf{a}\, \mathsf{True}\, b\, (\mathsf{Norm}\, s_0)$$

The first subgoal is proven by applying a Bali derived rule which says $\mathsf{initd}\, C\, s \Rightarrow G \vdash s - \mathsf{Init}\, C \rightarrow s$. If we look at the evaluation rule for the Init statement (5.6) and the equivalence $\mathsf{initd}\, C\, \sigma \equiv \mathsf{inited}\, C\, (\mathsf{globs}\, (\mathsf{store}\, \sigma))$ which we defined in Section 5.4.1, then immediately this derived rule can be seen to be true. The second subgoal essentially says that dynamic accessibility is not violated at runtime, *i.e.* the check_field_access function does not throw the error AccessViolation. This is where the five additional assumptions that we discussed earlier come into play, allowing us to apply the dynamic field access OK rule. We apply this rule, simplify and then we are done.

If the field is an instance field of an object, we repeat the same steps as above for the *stat* case, except when applying the evaluation rule for variable assignment, we instantiate w and f as follows:

$$\begin{aligned} w &\equiv \mathsf{the}\, (\mathsf{values}\, (\mathsf{the}\, (\mathsf{globs}\, s_0\, (\mathsf{Heap}\, (\mathsf{the\_Addr}\, b))))) (\mathsf{Inl}\, (\mathsf{a}, statDeclC))) \\ f &\equiv \lambda x.\, \mathsf{supd}\, (\mathsf{upd\_gobj}\, (\mathsf{Heap}\, (\mathsf{the\_Addr}\, b)) (\mathsf{Inl}\, (\mathsf{a}, statDeclC))\, x) \end{aligned}$$

We then apply the field variable evaluation rule and later the dynamic field access OK rule, instantiating appropriately and simplifying.

5.7 Array variable assignment

KeY's implementation of the array assignment rule is given as follows:

$$\langle \pi\, a[i] := val;\, \omega \rangle \phi \;\rightsquigarrow\; \begin{cases} a = null \rightarrow \langle NPE \rangle \langle \pi\, \omega \rangle \phi \\ (a \neq null \wedge (i < 0 \vee i \geq a.length)) \rightarrow \langle AOBE \rangle \langle \pi\, \omega \rangle \phi \\ (a \neq null \wedge i \geq 0 \wedge i < a.length) \rightarrow \{a[i] := val\} \langle \pi\, \omega \rangle \phi \end{cases}$$

Here a, i and val are side-effect free expressions. The formula on the left expresses that the assignment $a.[i] := val$ sequentially composed with ω terminates normally in a state in which the JavaCard DL formula ϕ holds. This is transformed into one of three formulae. The first is valid if a `NullPointerException` is thrown from a state satisfying $a = null$, *i.e* the reference that accesses a is `null`. The second formula is valid if an `ArrayIndexOutOfBounds` exception is thrown from a state satisfying $a \neq null$ and either $i < 0$ or i is greater or equal to the length of the array. The third formula is valid if from a state satisfying both $a \neq null$ and $0 \leq i < a.length$ in which $a.[i]$ has been updated by val, a run of the program $\pi\, \omega$ terminates in a state in which the formula ϕ holds.

We translate this directly into the following Bali rule.

$$\frac{\begin{array}{l} G \vdash \mathsf{Norm}\, s_0 - a \twoheadrightarrow b \rightarrow \mathsf{Norm}\, s_0 \\ G \vdash \mathsf{Norm}\, s_0 - i \twoheadrightarrow \mathsf{Intg}\, j \rightarrow \mathsf{Norm}\, s_0 \\ \mathsf{let}\, A = \mathsf{the\_Arr}\,(\mathsf{globs}\, s_0\,(\mathsf{Heap}\,(\mathsf{the\_Addr}\, b))); \\ \quad J = j\,\mathsf{in\_bounds}\,(\mathsf{fst}\,(\mathsf{snd}\, A)); \\ \quad U = \mathsf{upd\_gobj}\,(\mathsf{Heap}\,(\mathsf{the\_Addr}\, b))\,(\mathsf{Inr}\, j)\, c\, s_0 \\ \mathsf{in}\; G \vdash (\mathsf{raise\_if}\, \neg J\, \mathtt{IndOutBound}\,(\mathsf{np}\, b\, \mathsf{None}), s_0) - \mathit{val} \twoheadrightarrow c \rightarrow \mathsf{Norm}\, s_0; \\ G \vdash (\mathsf{if}\,(b = \mathsf{Null}) \\ \quad \mathsf{then}\,(\mathsf{Some}\,(\mathsf{StdXcpt}\,\mathtt{Nullpointer}), s_0) \\ \quad \mathsf{else}\,(\mathsf{if}\, \neg J\, \mathsf{then}\,(\mathsf{Some}\,(\mathsf{StdXcpt}\,\mathtt{IndOutBound}), s_0) \\ \qquad \mathsf{else}\,(\mathsf{None}, U))) - \omega \rightarrow \sigma_1 \end{array}}{G \vdash \mathsf{Norm}\, s_0 - (\mathsf{Expr}(a.[i] := \mathit{val})); \omega \rightarrow \sigma_1}$$

Expression a is first evaluated to value b. Since a is side-effect free, there is no state change brought about by the evaluation. Next, we evaluate the array index i to integer j. At the risk of causing two exceptions, we then evaluate *val* to the value c. If b is Null, a `Nullpointer` exception is thrown, otherwise a check is made to see whether j is within the bounds of the array. If it is out of bounds then a `IndOutBounds` exception is thrown.

This situation is again reproduced in the final assumption: if b is Null then the `Nullpointer` is propagated. If $b \neq \mathsf{Null}$ and j is out of bounds then the `IndOutBounds` exception is propagated. If j is within the bounds of the array and $b \neq \mathsf{Null}$ then the (global) store is updated. Finally the rest of the program ω is evaluated.

Unfortunately we cannot prove this rule correct. Recall that the function avar in the evaluation rule for array variables not only updates the state and checks for possible index bound violations, it also performs a dynamic type check on the value to be stored. If this type check fails an `ArrStore` exception is thrown. This models the Java scenario whereby an `ArrayStoreException` is thrown when an attempt is made to store the wrong type of object in an array. Hence in order to prove our rule correct we adjust our final assumption to incorporate this exception.

$$\begin{array}{l} G \vdash (\mathsf{if}\,(b = \mathsf{Null}) \\ \quad \mathsf{then}\,(\mathsf{Some}\,(\mathsf{StdXcpt}\,\mathtt{Nullpointer}), s_0) \\ \quad \mathsf{else}\,(\mathsf{if}\, \neg J\, \mathsf{then}\,(\mathsf{Some}\,(\mathsf{StdXcpt}\,\mathtt{IndOutBound}), s_0) \\ \qquad \mathsf{else}\,(\mathsf{if}\,(\neg G, s_0 \vdash c\,\mathsf{fits}\,(\mathsf{fst}\, A)) \\ \qquad\quad \mathsf{then}\,(\mathsf{Some}\,(\mathsf{StdXcpt}\,\mathtt{ArrStore}), s_0) \\ \qquad\quad \mathsf{else}\,(\mathsf{None}, U)))) - \omega \rightarrow \sigma_1 \end{array}$$

Now everything remains the same except the case whereby j is within the bounds of the array and $b \neq \mathsf{Null}$. When this occurs, a check is made to see whether c "fits" the type of the array. If c does not fit the type an `ArrStore` exception is thrown; if it does fit, the state (store) is updated accordingly.

The inability to prove our original translation implies that the possibility of a Java `ArrayStoreException` being thrown is missing from the KeY implementation of the array assignment rule. It is not sound for KeY to ignore the possibility of this exception. We suggest a predicate *typeof* be introduced to JavaCard DL, similar to the binary instance predicate *instanceof*. For example, suppose a *typeof* b is `true` if a and b are of the same type, or if the type of a

is a subtype of the type of b. The implementation of the array assignment rule could then be adapted as follows:

$$\langle \pi\, a[i] := val;\, \omega \rangle \phi \rightsquigarrow \begin{cases} a = null \rightarrow \langle NPE \rangle \langle \pi\, \omega \rangle \phi \\ (a \neq null \wedge (i < 0 \vee i \geq a.length)) \rightarrow \langle AOBE \rangle \langle \pi\, \omega \rangle \phi \\ (a \neq null \wedge i \geq 0 \wedge i < a.length \\ \qquad \wedge \neg (val\ typeof\ a)) \rightarrow \langle ASE \rangle \langle \pi\, \omega \rangle \phi \\ (a \neq null \wedge i \geq 0 \wedge i < a.length \\ \qquad \wedge val\ typeof\ a) \rightarrow \{a[i] := val\} \langle \pi\, \omega \rangle \phi \end{cases}$$

Proof of this rule *via* our Bali translation involves consideration of two cases. The case $b = \mathsf{Null}$ is trivially proven due to the presence of a `Nullpointer` exception in the initial state of our assumptions. The case $b \neq \mathsf{Null}$ is further subdivided into two cases where either j is within the bounds of the array or it is out of bounds. When j is out of bounds, we cause an `IndOutBound` exception to be present in an initial state of our assumption. When j is within the bounds of the array, we need to consider cases where c does or does not fit the array type. The case where c does not fit is trivially proven. For the case where c fits the array type, we apply the evaluation rule for sequential composition (5.1) to the conclusion. We then need to prove the following:

$$G \vdash \mathsf{Norm}\, s_0 - \mathsf{Expr}\, (a[i] := val) \rightarrow (\mathsf{None}, U)$$

We do this by first applying the expression evaluation rule (5.2) followed by the evaluation rule for the variable assignment expression (5.3), instantiating w and f by W and F which we define as follows:

$$\begin{aligned} W &\equiv \mathsf{the}\, (\mathsf{snd}\, (\mathsf{snd}\, A)(\mathsf{Inr}\, j)) \\ F &\equiv \lambda v (x, s').\, ((\mathsf{raise\_if}\, (\neg G, s' \vdash v\, \mathsf{fits}\, (\mathsf{fst}\, A))\, \texttt{ArrStore})\, x, U) \end{aligned}$$

After simplification we are left with the following goal.

$$G \vdash \mathsf{Norm}\, s_0 - a[i] \twoheadrightarrow (W, F) \rightarrow \mathsf{Norm}\, s_0$$

This can be proven by an application of the array variable evaluation rule (5.7), using the definition of the function avar and simplifying.

5.8 Conclusions and future work

We have translated a number of taclets – implementing the JavaCard DL assignment rules – into Bali and proven them correct with respect to this formalism. We have found one error in the KeY implementation of the array assignment rule and have suggested a possible solution. (This error has since been fixed.) Rather than "re-invent the wheel" by embedding the entire JavaCard DL semantics directly into a theorem prover, we have shown that it is possible to take advantage of an already established and embedded formalism such as Bali.

Whether it is a worthwhile exercise to translate all of KeY's taclets into Bali is debatable. As we have already seen in the translation of the field assignment rule, there are aspects of Bali (dynamic access, definite assignment) not covered by the JavaCard DL calculus. It is

also likely that not all KeY taclets can be translated into Bali (due to the presence of logical variables and abstract types). The assignment rules – being the only rules describing state changes – are crucial to the KeY implementation, hence it is of particular importance to formally prove their correctness. We advocate that the correctness of certain instances of taclets be considered, rather than aiming for overall correctness. Determining those taclets central to the KeY implementation and proving their correctness is the subject of future work. If this is done in combination with proof of correctness for derivable rules (as outlined in [23] where derived, non-axiomatic taclets for JavaCard DL are proven sound relative to the core set of JavaCard DL axioms) we move evermore closer towards proving the correctness of Java programs.

Chapter 6

Conclusion

Today computer programs are being written for a multitude of purposes and are being applied in an ever-increasing number of areas. Furthermore their complexity is increasing all the time. Faults within programs are commonplace: the result of typing mistakes, incorrect assumptions and vague, or even missing specifications. Such faults often have the potential to cause financial ruin or even endanger people's lives. Formal program verification has provided a means of restoring confidence in a program's correctness. Note that this process cannot *guarantee* correctness; a specification is merely a mathematical representation of a program's required properties; these can be incorrectly interpreted. However, with greater rigour applied during system development phases, developers are inclined to state requirements precisely and can more strongly argue the correctness of their implementation. In this thesis we have examined particular aspects of program verification in the context of the Java programming language. In the remainder, we summarise the contributions made and end with some concluding remarks.

Chapter 2 In this chapter we presented an extension of the Java Modeling Language, JML, with temporal logic. In the tradition of JML, the extension is designed to be simple, easy and intuitive to use for software engineers. We have described a semantics for our extension language and have shown how to translate a subset of the extension back into standard JML, thus allowing for the re-use of existing verification techniques. We have provided specifications for all temporal aspects of the JavaCard API using our extension. We have also shown how the language can be used to specify liveness properties, *i.e.* that eventually something good will happen, which currently cannot be specified in standard JML. Our work here has inspired Bellegarde, Groslambert *et al.* who have developed a method to verify liveness properties expressed using our extension language.

Chapter 3 Here we presented a method to factorise the verification of temporal properties for multi-threaded Java programs over their different threads. The method involves decomposing the program into different parts for which different verification tasks exist, hence providing added flexibility to program verification. Our property specification language, along with our program model, has been formalised in the theorem prover Isabelle/HOL. We have designed 25 rules – listed in Appendix A – that describe the factorisation of a given temporal property and have proven each rule correct with respect to our formalisation. In order to lighten the proof burden brought about by the state explosion problem, we advocate the combined use of our approach along with other techniques such as abstraction, slicing and atomicity checks.

Chapter 4 In this chapter we gave an overview of the different ways properties of relations (such as transitive closure, finiteness and generatedness) can be expressed in extensions of first-order logic (*i.e.* transitive closure logic, fixed-point logic and first-order dynamic logic). We discussed which of these extensions already are – or in fact should be – implemented within various object-oriented specification languages. Since the extensions of first-order logic are usually implemented within specification languages in a makeshift or *ad hoc* manner – most designers of specification languages are unaware of the logic underpinning their design decisions – it is a worthwhile exercise to investigate the means by which properties of relations can be expressed and how these means can be implemented within various languages.

Chapter 5 Here we translated a number of taclets – implementing the assignment rules of the JavaCard Dynamic Logic (DL) calculus within the KeY verification tool – into the Bali calculus and proved them correct with respect to the Bali operational semantics in Isabelle/HOL. We found one error in the KeY implementation of the array assignment rule and have suggested a possible solution. Rather than take a foundational approach, *i.e.* by embedding the JavaCard DL calculus within a theorem prover, we have shown that it is possible to take advantage of an already established and embedded formalism such as Bali in order to prove the correctness of an entirely independent verification tool.

In formal methods there is a compelling argument for the integration of both tools and methods. Formal deduction by interactive theorem proving is a highly effective form of program verification, but it requires significant amounts of time, effort and skill. On the other hand, a lightweight automatic model checker which does not require full functional specifications may be so "stripped down" that the verification it performs makes no guarantee that a particular property will hold over a program model. It is practical to use different tools that complement one another; large sections of less complicated specifications can be checked by automatic model checkers, whereas more precise verification methods – such as interactive theorem proving – can be applied to the more complex parts of the specifications. Moreover, given the unlikelihood of a single formal method being capable of describing and analysing every aspect of a complex system, it is desirable to use different verification methods in combination. When combining methods it is important however to take into consideration issues such as: semantic integration, specification language style and consistency, and the integration of proof systems. The techniques and methods presented throughout this thesis have been designed and developed with this in mind. For example, our extension of JML adds expressiveness to the specification language without hindering verification and simultaneously maintains the "spirit" of JML, *i.e.* its style and ease of use. Similarly, our factorisation method for the verification of temporal specifications can easily be used in conjunction with other methods such as abstraction and slicing without impacting upon these other approaches.

To conclude Formal methods enables developers to construct more reliable computational systems. However there is no single solution; there exists a multitude of approaches all with different goals, techniques and claims. Formal methods can be applied to all, or only selected components of a system. They can be applied with different levels of formality and at any stage of the development lifecycle. By building new tools, inventing new methods, integrating exist-

ing methods, and working with industry to transfer our methodologies effectively, it becomes more and more feasible to verify larger and increasingly complex computational systems.

Factorisation rules

A.1 Universality rules

$$\frac{\mathcal{T}_1 \models \mathsf{Universal}\, p\, \mathsf{Globally} \quad p \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_p\ \mathcal{T}_1 \mid \mathsf{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Universal}\, p\, \mathsf{Globally}}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Universal}\, p\, \mathsf{After}\, q \quad p \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_p\ \mathcal{T}_1 \mid \mathsf{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Universal}\, p\, \mathsf{After}\, q}$$

$$\frac{\mathcal{T}_1 \models \mathsf{Universal}\, p\, \mathsf{Before}\, r \quad \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid \mathsf{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Universal}\, p\, \mathsf{Before}\, r}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Universal}\, p\, \mathsf{Between}\, q\, r \quad \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid \mathsf{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Universal}\, p\, \mathsf{Between}\, q\, r}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Universal}\, p\, \mathsf{AfterUntil}\, q\, r \quad p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Universal}\, p\, \mathsf{AfterUntil}\, q\, r}$$

A.2 Existence rules

$$\frac{\mathcal{T}_1 \models \mathsf{Exists}\, p\, \mathsf{Globally} \quad \neg p \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_p\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{Globally}}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Exists}\, p\, \mathsf{After}\, q \quad \neg p \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_p\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{After}\, q}$$

$$\frac{\mathcal{T}_1 \models \mathsf{Exists}\, p\, \mathsf{Before}\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{Before}\, r}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{Between}\, q\, r}$$

$$\frac{\mathcal{T}_1, q \models \mathsf{Exists}\, p\, \mathsf{AfterUntil}\, q\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ \mathsf{dep}_{p,r}\ \mathcal{T}_1 \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \mathsf{Exists}\, p\, \mathsf{AfterUntil}\, q\, r}$$

A.3 Precedence rules

$$\frac{\mathcal{T}_1 \models p \,\mathsf{Precedes}\, q \,\mathsf{Globally} \quad \neg p \wedge \neg q \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{Precedes}\, q \,\mathsf{Globally}}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{Precedes}\, q \,\mathsf{After}\, r \quad \neg p \wedge \neg q \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{Precedes}\, q \,\mathsf{After}\, r}$$

$$\frac{\mathcal{T}_1 \models p \,\mathsf{Precedes}\, q \,\mathsf{Before}\, r \quad \neg p \wedge \neg q \wedge \neg r \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,r}\, \mathcal{T}_1 \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{Precedes}\, q \,\mathsf{Before}\, r}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{Precedes}\, q \,\mathsf{Between}\, r\, s \quad \neg p \wedge \neg q \wedge \neg s \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,s}\, \mathcal{T}_1 \mid p \vee s}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{Precedes}\, q \,\mathsf{Between}\, r\, s}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{Precedes}\, q \,\mathsf{AfterUntil}\, r\, s \quad \neg p \wedge \neg q \wedge \neg s \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,s}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{Precedes}\, q \,\mathsf{AfterUntil}\, r\, s}$$

A.4 Response rules

$$\frac{\mathcal{T}_1 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Globally} \quad \neg p \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Globally}}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{RespondsTo}\, q \,\mathsf{After}\, r \quad \neg p \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{After}\, r}$$

$$\frac{\mathcal{T}_1 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Before}\, r \quad \neg p \wedge \neg r \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,r}\, \mathcal{T}_1 \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Before}\, r}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Between}\, r\, s \quad \neg p \wedge \neg s \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,s}\, \mathcal{T}_1 \mid p \vee s}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{Between}\, r\, s}$$

$$\frac{\mathcal{T}_1, r \models p \,\mathsf{RespondsTo}\, q \,\mathsf{AfterUntil}\, r\, s \quad \neg p \wedge \neg s \models \mathcal{T}_2 \,\mathsf{preserves}\, \mathsf{dep}_{p,q,s}\, \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models p \,\mathsf{RespondsTo}\, q \,\mathsf{AfterUntil}\, r\, s}$$

Bibliography

[1] J-R. Abrial. *The B Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P.H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] N. Alechina, S. Demri, and M. de Rijke. A modal perspective on path constraints. *Journal of Logic and Computation*, 13:1–18, 2003.

[4] N. Alechina and N. Immerman. Reachability logic: an efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337, 2000.

[5] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[6] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. The Java Series. Addison Wesley, 3rd edition, 2000.

[7] T. Baar. The definition of transitive closure with OCL - limitations and applications. In *Proceedings of 5th Andrei Ershov International Conference on Perspectives of System Informatics*, number 2890 in LNCS, pages 358–365. Springer, 2003.

[8] Project Bali. `http://isabelle.in.tum.de/bali/`.

[9] The Bandera Project. `http://bandera.projects.cis.ksu.edu/`.

[10] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *Runtime Verification*, volume 55(2) of *ENTCS*. Elsevier, 2001.

[11] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, number 2041 in LNCS, pages 6–24. Springer, 2001.

[12] B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1):1–36, 2004.

[13] B. Beckert and P.H. Schmitt. Program verification using change information. In *Software Engineering and Formal Methods (SEFM 2003)*, pages 91–99. IEEE Press, 2003.

[14] B. Beckert and K. Trentelman. Second-order principles in specification languages for object-oriented programs. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2005)*, number 3835 in LNCS, pages 154–168. Springer, 2005.

[15] F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical Report RR-5331, INRIA Sophia Antipolis, 2004.

[16] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in LNCS, pages 299–312. Springer, 2001.

[17] Y. Bertot and P. Castéran. *Coq'Art: the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[18] M. Bidoit and P.D. Mosses. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*. Number 2900 in LNCS. Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[19] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 353–404. Springer, 1999.

[20] L. du Bousquet, J-L. Lanet, Y. Ledru, O. Maury, and C. Oriat. A case study in JML-based software validation. In *Automated Software Engineering (ASE 2004)*, pages 294–297. IEEE Press, 2004.

[21] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. Technical Report NIII-R0316, NIII, University of Nijmegen, 2004.

[22] C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST 2002)*, number 2422 in LNCS, pages 304–318. Springer, 2002.

[23] R. Bubel, A. Roth, and P. Rümmer. Ensuring correctness of lightweight tactics for Java Card Dynamic Logic. In *Proceedings of Workshop on Logical Frameworks and Meta-Languages at IJCAR 2004*, pages 84–105, 2004.

[24] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.

[25] L. Burdy, A. Requet, and J-L. Lanet. Java applet correctness: a developer oriented approach. In *Formal Methods Europe (FME 2003)*, number 2805 in LNCS, pages 422–439. Springer, 2003.

[26] N. Cataño. *Formal Methods for Java Programs*. PhD thesis, Université de Paris 7 and INRIA, France, 2004.

[27] N. Cataño and M. Huisman. Formal specification and static checking of Gemplus's electronic purse using ESC/Java. In L-H. Eriksson and P.A. Lindsay, editors, *Formal Methods Europe (FME 2002)*, number 2391 in LNCS, pages 272–289. Springer, 2002.

[28] N. Cataño and M. Huisman. Chase: a static checker for JML's assignable clause. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Check-*

ing and Abstract Interpretation (VMCAI 2003), number 2575 in LNCS, pages 26–40. Springer, 2003.

[29] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, pages 175–200. Springer, 1999.

[30] M.V. Cengarle and A. Knapp. A formal semantics for OCL 1.4. In *UML 2001 - the Unified Modeling Language, Modeling Languages, Concepts, and Tools*, number 2185 in LNCS, pages 118–133. Springer, 2001.

[31] H. Chen, J. Hsiang, and H.C. Kong. On finite representation of infinite sequences of terms. In *Proceedings of 2nd International Workshop on Conditional and Typed Rewriting Systems*, number 516 in LNCS, pages 100–114. Springer, 1990.

[32] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison Wesley, 2000.

[33] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[34] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Transactions on Programming Languages and Systems (TOPLAS 1994)*, volume 16(5), pages 1512–1542. ACM Press, 1994.

[35] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of 4th Symposium on Logic in Computer Science*, pages 353–362, 1989.

[36] E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[37] CoFI. `http://www.cofi.info/`.

[38] D.R. Cok and J. Kiniry. ESC/Java2: uniting ESC/Java and JML – progress and issues in building and using ESC/Java2. In G. Barthe, L. Burdy, M. Huisman, J-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, number 3362 in LNCS, pages 108–128. Springer, 2004.

[39] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL, 1999. `http://www.trireme.com/whitepapers/design/components/OCL\_manifesto.PDF`.

[40] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-states models from Java source code. In M. Jazayeri and A. Wolf, editors, *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, 2000.

[41] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the Bandera specification language. Technical Report 2001-04, Kansas State University, Computing and Information Sciences, 2001.

[42] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, pages 313–400. World Scientific, 1997.

[43] The Daikon Invariant Detector. http://pag.csail.mit.edu/daikon/.

[44] A. Dawar and Y. Gurevich. Fixed point logics. In *The Bulletin of Symbolic Logic*, volume 8, pages 65–88. Association for Symbolic Logic, 2002.

[45] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, Hewlett Packard Systems Research Center, 2003.

[46] M.B. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, pages 7–15, 1998.

[47] M.B. Dwyer, J. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng. Slicing multi-threaded Java programs: a case study. Technical Report 99-7, Kansas State University, Computing and Information Sciences, 1999.

[48] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.

[49] C. Engel. A Translation from JML to Java Dynamic Logic. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2005.

[50] C. Engel and A. Roth. KeY Quicktour for JML. http://www.key-project.org/download/.

[51] ESC/Java2. http://www.cs.kun.nl/sos/research/escjava/index.html.

[52] C. Flanagan and S.N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI 2000)*, pages 219–232, 2000.

[53] C. Flanagan and S.N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In X. Leroy, editor, *Principles of Programming Languages (POPL 2004)*, pages 256–267, 2004.

[54] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI 2002)*, pages 234–245, 2002.

[55] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation (PLDI 2003)*, pages 338–349, 2003.

[56] M. Fowler and K. Scott. *UML Distilled: a Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2nd edition, 2000.

[57] S.N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[58] Gemplus. http://www.gemplus.com/.

[59] A. Giorgetti and J. Groslambert. JAG: JML annotation generation for verifying temporal properties, 2005. http://lifc.univ-fcomte.fr/~groslambert/JAG/.

[60] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes. Stanford University, 2nd edition, 1992.

[61] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, 2nd edition, 2000.

[62] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, 1984.

[63] J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis (SAS 1999)*, number 1694 in LNCS, pages 1–18. Springer, 1999.

[64] J. Hatcliff, M.B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[65] J. Hatcliff, Robby, and M.B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In B. Steffen and G. Levi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, number 2937 in LNCS, pages 175–190. Springer, 2004.

[66] K. Havelund and G. Roşu. Java PathExplorer – a runtime verification tool. In *Proceedings of 6th International Symposium on AI, Robotics and Automation in Space*, 2001.

[67] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[68] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[69] M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[70] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE 2000)*, number 1783 in LNCS, pages 284–303. Springer, 2000.

[71] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's vector class. *Software Tools for Technology Transfer*, 3(3):332–352, 2001.

[72] M. Huisman and K. Trentelman. Factorising temporal specifications. Technical Report RR-5326, INRIA, 2004.

[73] M. Huisman and K. Trentelman. Factorising temporal specifications. In M. Atkinson and F. Dehne, editors, *Proceedings of Computing: the Australasian Theory Symposium (CATS 2005)*, pages 87–96. Australian Computer Society, 2005.

[74] M.R.A. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[75] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[76] Isabelle. `http://isabelle.in.tum.de/index.html`.

[77] D. Jackson. Automating first-order relational logic. In *Foundations of Software Engineering*, pages 130–139, 2000.

[78] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of 22nd International Conference on Software Engineering*, pages 730–733. ACM Press, 2000.

[79] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[80] B. Jacobs, C. Marche, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST 2004)*, number 3116 in LNCS, pages 21–22. Springer, 2004.

[81] B. Jacobs, H. Meijer, and E. Poll. VerifiCard: a European project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001.

[82] B. Jacobs, M. Oostdijk, and M. Warnier. Source code verification of a secure payment applet. *Journal of Logic and Algebraic Programming*, 58:107–120, 2004.

[83] B. Jacobs and E. Poll. A logic for the Java modeling language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE 2001)*, number 2029 in LNCS, pages 284–299. Springer, 2001.

[84] B. Jacobs and E. Poll. Java program verification: developments and perspective. Technical Report NII-R0318, Nijmegen Institute for Computing and Information Sciences, 2003.

[85] J. Jansen. Slicing Midlets. Master's thesis, University of Nijmegen, 2004.

[86] The Jass Page. `http://csd.informatik.uni-oldenburg.de/~jass/`.

[87] Java PathFinder. `http://ase.arc.nasa.gov/havelund/jpf.html`.

[88] JavaCard 2.1. API. `http://java.sun.com/products/javacard/htmldoc/`.

[89] JavaCard Technology. `http://java.sun.com/products/javacard/`.

[90] JML Specifications for JavaCard's API. `http://www.sos.cs.ru.nl/research/escjava/esc2jcapi.html`.

[91] The KeY Project. `http://www.key-project.org/`.

[92] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990.

[93] Krakatoa. `http://krakatoa.lri.fr/`.

[94] S. Kreutzer. Expressive equivalence of least and inflationary fixed-point logic. In *Proceedings of Symposium on Logic in Computer Science*, page 403. IEEE, 2000.

[95] S. Kreutzer. *Pure and Applied Fixed-Point Logics*. PhD thesis, Aachen University of Technology, 2002.

[96] K.R.M. Leino. Specifying the modification of extended state. Technical Report 1997-026, Digital Systems Research Center, 1997.

[97] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 2000.

[98] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106. ACM Press, 2000.

[99] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual. `http://www.cs.iastate.edu/~leavens/JML//jmlrefman/`.

[100] K.R.M. Leino, G. Nelson, and J.B. Saxe. ESC/Java user's manual. Technical Report 2000-02, Compaq System Research Center, 2000.

[101] L. Mandel and M.V. Cengarle. On the expressive power of OCL. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, number 1708 in LNCS, pages 854–874. Springer, 1999.

[102] C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[103] R. Mateescu. Local model-checking of an alternation-free value-based modal mu-calculus. In A. Bossi, A. Cortesi, and F. Levi, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI 1998)*, 1998.

[104] H. Meijer and E. Poll. Towards a full formal specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security (E-smart 2001)*, number 2140 in LNCS, pages 165–178. Springer, 2001.

[105] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[106] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[107] P.D. Mosses. CASL: a guided tour. `http://www.brics.dk/Projects/CoFI/Documents/CASL/GuidedTour/index.html`.

[108] P. Müller, A. Poetzsch-Heffter, and G.T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02, Iowa State University, Department of Computer Science, 1997.

[109] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL – a Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

[110] OCL Center. `http://www.klasse.nl/ocl/`.

[111] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[112] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[113] D. von Oheimb and T. Nipkow. *Machine-Checking the Java Specification: Proving Type Safety*, chapter 4, pages 119–156. Number 1523 in LNCS. Springer, 1999.

[114] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM Press, 1984.

[115] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV 1996)*, number 1102 in LNCS, pages 411–414. Springer, 1996.

[116] L.C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in LNCS. Springer, 1994.

[117] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Applications (CARDIS 2000)*, pages 135–154. Kluwer, 2000.

[118] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[119] I. Prasetya and S. Swierstra. Factorizing fault tolerance. *Theoretical Computer Science*, 290(2):1201–1222, 2003.

[120] I. Prasetya, S. Swierstra, and B. Widjaja. Component-wise formal approach to design distributed systems. `http://citeseer.ist.psu.edu/prasetya99componentwise.html`, 1999.

[121] Proof General. `http://proofgeneral.inf.ed.ac.uk/`.

[122] A.D. Raghavan and G.T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, Department of Computer Science, 2000.

[123] Robby. Bandera Specification Language: a Specification Language for Software Model Checking. Master's thesis, Kansas State University, 2000.

[124] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA 1987)*, pages 466–481, 1987.

[125] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1, SRI International, 1995.

[126] J. Rushby. Theorem proving for verification. In *Modeling and verification of parallel processes*, number 2067 in LNCS, pages 39–57. Springer, 2001.

[127] A. Santone. Automatic verification of concurrent systems using a formula-based compositional approach. *Acta Informatica*, 38:531–564, 2002.

[128] B. Sasse. Formal Correctness of a Program Logic Calculus for the Deductive Verification of Java Programs. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, 2002.

[129] N. Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 2003.

[130] N. Schirmer. Java definite assignment in Isabelle/HOL. In *Proceedings of ECOOP Workshop on Formal Techniques for Java-like Programs*, 2003.

[131] A. Schürr. Adding graph transformation concepts to UML's constraint language OCL. In H. Ehrig, C. Ermel, and J. Padberg, editors, *Uniform Approaches to Graphical Process Specification Techniques*, volume 44(4) of *ENTCS*. Elsevier, 2001.

[132] Security of Systems Group. `http://www.cs.kun.nl/sos/index.html`.

[133] Specification Patterns Project. `http://patterns.projects.cis.ksu.edu/`.

[134] Spin. `http://spinroot.com/spin/whatispin.html`.

[135] F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In G. Ghelli, editor, *Foundations of Object-Oriented Languages (FOOL-10)*, 2003.

[136] The SQL Page. `http://www.jcc.com/SQLPages/jccs\_sql.htm`.

[137] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1995.

[138] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[139] K. Trentelman. Proving correctness of JavaCard DL taclets using Bali. To be published in the Proceedings of the International Conference on Software Engineering and Formal Methods, 2005.

[140] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST 2002)*, number 2422 in LNCS, pages 334–348. Springer, 2002.

[141] UML 1.3 Specification. `http://www.omg.org`.

[142] The Verificard Project. `http://www.verificard.org/`.

[143] F. Wang. Formal verification of timed systems: a survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, 2004.

[144] M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, 1979.

[145] The Why Tool. `http://why.lri.fr/`.

[146] Wikipedia. `http://en.wikipedia.org/`.

[147] J.M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.

[148] J. Yang. SQL3 recursion. Lecture Notes, Stanford University, 1999.

[149] S. Zilles. Algebraic specification of data types. Technical Report XI, MIT Laboratory for Computer Science, 1974.

Index